SINCRONIZACIÓN PARALELA DE TRAYECTORIAS PARA DETECCIÓN DE CONFLICTOS ENTRE AERONAVES

EDUARDO DE LA IGLESIA RAMÍREZ

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería de Computadores

Junio de 2013

Director:
GUILLERMO BOTELLA JUAN
Director Colaborador externo:
CARLOS GARCÍA SÁNCHEZ

Autorización de Difusión

EDUARDO DE LA IGLESIA RAMÍREZ

Junio de 2013

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Sincronización paralela de trayectorias para detección de conflictos entre aeronaves”, realizado durante el curso académico 2012-2013 bajo la dirección de Guillermo Botella Juan y Carlos García Sánchez en el Departamento de Ingeniería de Computadores, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

El transporte aéreo es un sector clave para la sociedad y la economía. Los sistemas de control de tráfico aéreo tienen la necesidad de mejorar sus capacidades y contribuir al desarrollo de nuevas tecnologías ante la cada vez mayor demanda del espacio aéreo. Las complejas rutas que siguen las aeronaves en el cielo necesitan ser monitorizadas y alteradas en tiempo real por diversas razones, como la eficiencia en las operaciones, la seguridad y la sostenibilidad.

El objetivo de este trabajo es un estudio de viabilidad del problema real de sincronización paralela de trayectorias en aviónica. En este contexto, las unidades de procesamiento gráfico (GPUs) son una poderosa herramienta que puede permitir mejorar estos algoritmos en términos de eficiencia y rendimiento. Tratamos de identificar cuándo el uso de un algoritmo paralelo resulta beneficioso con respecto a uno equivalente que resuelve el problema de forma secuencial y cómo se adaptan dichos algoritmos para ser ejecutados en una GPU.

El trabajo parte de un estado del arte actual, haciendo uso de conceptos validados por expertos dentro del marco de SESAR (Single European Sky ATM Research), ambicioso programa de investigación y desarrollo llevado a cabo por la Unión Europea, EUROCONTROL y las más relevantes empresas privadas de la industria aeronáutica. Los conceptos usados principalmente son: Extended Projected Profile (EPP), Air Ground Data Link (AGDL) y Automatic Dependent Surveillance - Contract (ADS-C), gracias a los cuáles se puede obtener una trayectoria estimada por la aeronave en tiempo real y usarla como base de nuestra investigación.

Palabras clave

Trayectoria de Aeronave, EPP, Control de Tráfico Aéreo, AGDL, ADS-C, GPU, Sincronización de Trayectorias, Resolución de Conflictos, CUDA.

Resumen en inglés

Air transport is a key sector for our society and economy. Air Traffic Control Systems should improve their capabilities and contribute to development of new technologies to the increasing demand of the airspace. The complex routes followed by the aircrafts in the sky need to be monitored and changed in real time for various reasons such as operational efficiency, safety and sustainability.

The aim of this work is to investigate the use of parallel algorithms for comparing aircraft trajectories. In this context, graphics processing units (GPUs) are a powerful tool that could improve these algorithms in efficiency and performance terms. We try to identify when the use of the parallel capacities in novel hardware devices are more beneficial to aim in Air Traffic Control Systems.

This project starts from the last Air Traffic Control state of the art, using concepts validated by experts within the framework of SESAR (Single European Sky ATM Research), ambitious research and development program undertaken by the European Union, EUROCONTROL and the most relevant companies of aviation industry. Due to use of the following concepts: Extended Projected Profile (EPP), Air Ground Data Link (AGDL) and Automatic Dependent Surveillance - Contract (ADS-C), we can obtain an estimated trajectory from the aircraft in real time and use it as our base of researching.

Keywords

Aircraft Trajectory, EPP, Air Traffic Control, AGDL, ADS-C, GPU, Trajectory Synchronization, Conflict Resolution, CUDA.

Índice de contenidos

Índice de figuras

Índice de tablas

Agradecimientos

Toda mi gratitud a Mª Carmen por su constante apoyo y sus motivadoras palabras que me han empujado a dedicar todo mi esfuerzo a este Máster en Investigación.

A mis padres por su cariño y por darme la posibilidad de conocer y estudiar esta ciencia que tanto me gusta.

A mis profesores directores por sus indicaciones y consejos, de gran utilidad para enfocar el contenido y el objetivo de este trabajo.

1 - Introducción

La tarea de comparar múltiples trayectorias es un problema bastante costoso. Para conseguir la trayectoria de una aeronave suelen usarse predictores de trayectoria - Trajectory Predictor (TP) en inglés - que tratan de modelar el recorrido de un avión y estimar su trayectoria en un intervalo de tiempo futuro. Un TP debe de tomar una gran cantidad de datos de entrada para afinar la predicción lo más posible. De forma tradicional han sido implementados modelos de predicción [10] en los centros de control aéreo empleando información proporcionada por los radares, con los datos previos de planes de vuelo, evaluando la congestión del tráfico aéreo, la información meteorológica y las características físicas de cada modelo determinado de avión con el ánimo de prevenir riesgos.

En los últimos años se han hecho bastantes esfuerzos por facilitar la comunicación entre los centros de control aéreo y las aeronaves. Un ejemplo de ello es el uso de AGDL (Air Ground Data Link), que establece un canal de comunicación entre las propias aeronaves y los controladores aéreos [11]. Entre sus Aplicaciones se encuentra ADS (Automatic Dependent Surveillance) que permite a los aviones proporcionar automáticamente datos derivados de la navegación a bordo y de su estado (identificación, posición en cuatro dimensiones y datos adicionales) a los centros de control aéreo.

En este trabajo se tomarán como entrada las trayectorias predichas por cada aeronave y obtenidas en tierra por medio de la Aplicación ADS. Cabe destacar que la propia aviónica debería poder hacer predicciones de su trayectoria más exactas de lo que un sistema en tierra sería capaz de calcular, pues conoce de primera mano el estado actual (posición, velocidad, aceleración, cantidad de combustible, parámetros de vuelo, etc.) y modelo de comportamiento y actuación del avión concreto.

A continuación, en la Figura 1.1, se puede ver un ejemplo de trayectoria trazada para el último tramo de una aeronave con destino al aeropuerto de Madrid-Barajas.

Figura 1.1 Trayectoria de una aeronave en aproximación a Madrid-Barajas

Una vez obtenida la predicción de trayectoria de varias aeronaves se puede utilizar para distintos fines. Un uso muy común es la comparación de la trayectoria obtenida por medio de ADS y la progresión de la aeronave que se tiene prevista en tierra para que el controlador pueda dar instrucciones válidas en su gestión del espacio aéreo como se refleja en [3] y [6]. Otra posible aplicación es la de comparar múltiples trayectorias de aeronaves para detectar posibles conflictos y permitir a los controladores su resolución, problema estudiado en [5], [8] y [13]. Debido a la complejidad de las trayectorias que se reciben por ADS, un algoritmo de comparación secuencial resulta costoso y poco eficiente. Además, en este contexto de tiempo real, el tiempo de respuesta de este algoritmo es crucial. Si vamos a detectar un conflicto entre dos aeronaves dentro de unos pocos minutos, el algoritmo de comparación no puede tardar demasiado en avisar de este conflicto al controlador porque se pondría en riesgo la seguridad de las aeronaves y se violaría la distancia de separación entre las mismas.

El objetivo de este trabajo es desarrollar una metodología mediante el uso de las capacidades computacionales disponibles hoy en día con los dispositivos GPU que posibilite la comparación entre trayectorias recibidas por el sistema en tierra para detectar posibles conflictos entre aeronaves de forma eficiente y segura, y mejore el orden y la rapidez de las operaciones aéreas. En este trabajo se explorarán las ventajas de los aceleradores gráficos (GPUs) que permiten explotar el paralelismo gracias a sus múltiples núcleos posibilitando el lanzamiento de un altísimo número de hilos simultáneos, como se ha investigado recientemente en este campo

[2] en el contexto de la Gestión de Tráfico Aéreo. Además, se abordará la problemática en la explotación eficiente de la jerarquía de memoria en estos aceleradores haciendo uso de patrones generales de acceso para la resolución de problemas paralelos.

1.1 - Control de Tráfico Aéreo

El Control de Tráfico Aéreo (Air Traffic Control, ATC) es un servicio reglamentado cuya labor consiste en distribuir el Espacio Aéreo controlando la sepación de las aeronaves que pretenden utilizarlo. Este servicio se realiza, principalmente, por controladores ubicados en los centros de operaciones y torres de aeródromos que se encuentran comunicados entre sí y con los pilotos de las aeronaves.

Para desemplear el servicio de Control de Tráfico Aéreo se tiene el apoyo de la información suministrada por diversos dispositivos electrónicos, sistemas informáticos y de comunicaciones que interpretada y gestionada por el controlador es aceptada por el piloto de la aeronave.

Las prioridades del servicio son garantizar la seguridad y proporcionar orden y rapidez al tráfico aéreo. El intercambio de información hablada entre el personal de tierra y el piloto ha sido históricamente el método más utilizado, crucial para garantizar los movimientos de la aeronave y la distancia de separación con el resto. Este canal de comunicación se establece vía radio en el rango de las frecuencias VHF (UHF de forma exclusiva en ámbito militar). El contenido del mensaje hablado emitido y su respuesta se rige por unas normas de lenguaje muy estrictas que conforman la fraseología aeronáutica y en uno de los idiomas autorizados por la OACI (ICAO - International Civil Aviation Organization).

El uso habitual de Radares para la localización de las aeronaves ha sido también un elemento de gran importancia en el Control de Tráfico Aéreo, tan esencial como las comunicaciones por voz.

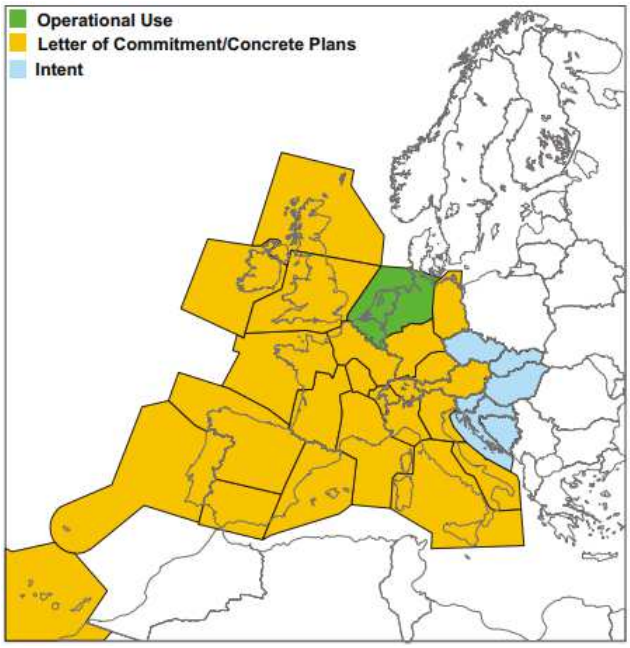
Ante el aumento exponencial del tráfico aéreo, los organismos internacionales alertaron en el pasado sobre las limitaciones de ambos sistemas. Entre otras, se presentan problemas relacionados con la congestión de frecuencias de radio, las interferencias en la comunicación por voz tanto limitadas por coberturas de radar debido a la orografía del terreno, reflexiones y ecos falsos, como por el control oceánico sin cobertura radar. Es por ello, que ICAO propone un

nuevo concepto de control aéreo llamado CNS/ATN (Communications, Navigation and Surveillance / Air Traffic Management) que complemente, y sustituya poco a poco el método anterior, como se detalla en [18]. La intercomunicabilidad entre las redes de telecomunicaciones aeronáuticas hace que aparezca un nuevo protocolo estándar para los enlaces de datos que se comenta a continuación.

1.2 - Air Ground Data Link

Air Ground Data Link, en adelante AGDL, establece un enlace de datos entre el centro de control de tráfico aéreo y la aeronave. Está regulado internacionalmente por ICAO y Eurocae, existiendo además una regulación publicada por la Comisión Europea para las reglas de implementación de los servicios AGDL. Se encuentra en uso operacional en varios de los espacios aéreos con mayor tráfico e importancia de Europa y deberá ser introducido de manera continua y homogénea en el resto de zonas para completar el Cielo Único Europeo, como se muestra en la Figura 1.2.

Figura 1.2 Servicios AGDL en los FIR Europeos en 2011



Se espera que los FIR de la zona en amarillo usen AGDL de forma operacional durante este año (2013) y el resto de regiones europeas lo hagan a partir de 2015.

Las principales aplicaciones y servicios AGDL son los siguientes:

Aplicación CM (Context Management)

- DLIC (Data Link Initiation Capability service). Permite el establecimiento del enlace entre los sistemas de tierra y aire. Asocia la aeronave con un plan de vuelo y permite el intercambio de datos para el uso del resto de servicios.

Aplicación CPDLC (Controller Pilot Data Link Communications)

- ACM (ATC Communications Management service). Proporciona la transferencia automatizada de las comunicaciones datalink y radio entre sectores y centros de control.
- ACL (ATC Clearances and Information service). Proporciona la capacidad de intercambio de mensajes operacionales, tales como las peticiones e informes desde la aeronave como los mandatos, instrucciones y notificaciones por parte del controlador.
- AMC (ATC Microphone Check service). Permite al controlador enviar la instrucción de chequeo de micrófono a todas las aeronaves bajo su control para verificar que sus canales de voz no están bloqueados.

Existen otros servicios asociados a CPDLC que no deben ser implementados de forma obligatoria, a diferencia de los descritos aquí, que carecen de interés para nuestro objetivo en este trabajo.

Aplicación ADS (Automatic Dependent Surveillance)

Permite al sistema de tierra establecer un contrato mediante ADS-C (ADS-Contract) con la aeronave de forma que ésta obtenga la información de sus sensores de abordo y los envíe de forma periódica, por evento o bajo demanda. El contrato es un acuerdo dinámico entre ambos sistemas, y será la base del algoritmo para obtener las trayectorias. ADS-C, definida por el

estándar AGDL, está en fase experimental. Existen otras modalidades como ADS-B (ADS-Broadcast) que están empezando también a ser usadas en otros lugares como Estados Unidos.

1.3 - Objetivos del trabajo

El principal objetivo de este trabajo es realizar un estudio de viabilidad -en cuanto a aceleración se refiere- de sincronización paralela de trayectorias para detección de colisiones en aeronaves, usando hardware gráfico. Se pretende evaluar la conveniencia de usar dispositivos GPU con este propósito.

Si bien a día de hoy las trayectorias empleadas en aviónica se encuentran simplificadas con un número de puntos reducido para que su procesado no sea costoso y por ende se puedan estimar posibles colisiones en tiempo real, en un futuro no muy lejano se tienden a emplear trayectorias más realistas que conlleven mayores necesidades computacionales. Bajo esta premisa en este trabajo se abordará un estudio relativo a las posibilidades que ofrece la tecnología basada en GPU cuyo modelo de programación difiere substancialmente al que estamos acostumbrados en un procesador de propósito general tipo CPU pero que ofrece unas tasas de rendimiento (número de operaciones por segundo) mucho más elevadas.

Además, la metodología propuesta a lo largo del trabajo posibilita la construcción de un sistema tolerante a fallos por medio del que se puedan obtener las estimaciones precisas de colisiones con bajos tiempos de respuesta. Esta metodología facilita la verificación y validación de los resultados.

Se pretende también adquirir el conocimiento necesario para poder desarrollar futuras investigaciones de aceleración hardware en el contexto de la aviación comercial y el control de tráfico aéreo [1] .

2 - Unidad de Procesamiento Gráfico

Una unidad de procesamiento gráfico (GPU) es un microprocesador dedicado específicamente al renderizado de gráficos. Su labor es el procesamiento de una descripción de alto nivel de una escena para obtener una imagen bidimensional rasterizada, liberando a la CPU de gran parte de la carga de trabajo del proceso y acelerándolo significativamente. Las modernas GPUs son descendientes de los chips gráficos monolíticos de finales de la década de 1970 y 1980. A partir de los años 90, los microprocesadores de propósito general de alta velocidad fueron muy populares para implementar las GPUs más avanzadas. Por ejemplo, muchas impresoras laser contenían un procesador de barrido de imágenes basado en un procesador RISC. Conforme la tecnología de proceso de semiconductores fue mejorando se hizo posible trasladar estos aceleradores gráficos al mismo chip, siendo más fáciles de hacer y vender.

En los últimos años el desarrollo de las GPU ha sido fuertemente impulsado por la industria de los videojuegos y multimedia, y como resultado hoy en día están ampliamente disponibles en estaciones de trabajo, ordenadores personales, smartphones y tablets. Así pues, existen tarjetas gráficas comerciales como las actuales unidades de procesamiento gráfico de NVIDIA y ATI que disponen de una interfaz de programación que permite utilizarlas para tareas de propósito general.

Las capacidades computacionales de este tipo de aceleradores ha llamado poderosamente la atención en la comunidad científica, consolidándose como una tecnología madura. En la actualidad, y como podemos comprobar en [23], los computadores más potentes del mundo basan su rendimiento pico en aceleradores gráficos, siendo el único camino para seguir incrementando las prestaciones computaciones, y hacer que la ley de Moore siga teniendo validez. En este trabajo se explorará la posibilidad de emplear este tipo de dispositivos en el ámbito de la comparación de trayectorias y la evaluación de colisiones aéreas siendo esta tarea lo suficientemente pesada como para no alcanzar los requisitos de tiempo real en sistemas computacionales convencionales, por lo que pueden ser aprovechadas en pesados cálculos que los algoritmos de comparación de trayectorias computan.

CUDA es la arquitectura que permite codificar algoritmos para GPU de NVIDIA siendo ésta una variación del lenguaje de programación C. Sin embargo, CUDA es una arquitectura propietaria únicamente disponible en los sistemas de NVIDIA. OpenCL nace como una interfaz

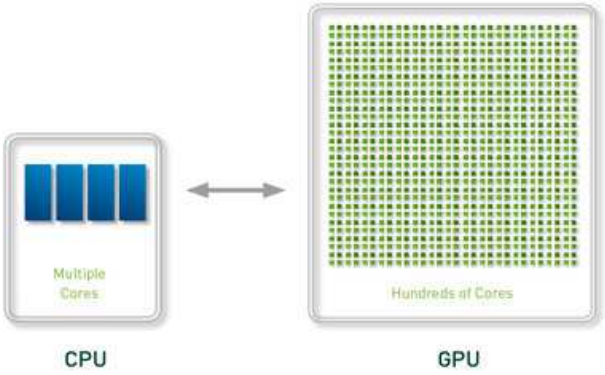
de programación abierta que permite crear algoritmos con paralelismo a nivel de datos y de tareas que se pueden ejecutar tanto en CPU como en GPU, y siendo más generalista, pudiendo ser utilizado entre otras en tarjetas gráficas de NVIDIA, ATI o Intel.

2.1 - Programación de propósito general sobre GPUs

La ventaja principal de utilizar las GPUs en aplicaciones de propósito general es que sus capacidades computacionales mejoran la eficiencia y reducen el consumo tanto de recursos como de tiempo en tareas que son altamente paralelizables. En el apartado concreto de este trabajo como caso de estudio de la comparación de trayectorias, tenemos como objetivo explotar el paralelismo en aquellos cálculos que sean susceptibles de ser paralelizados.

Otra ventaja relativa al uso de GPUs, es que en los próximos años la tendencia hace indicar que seguirá incrementándose el número de procesadores/cores en las tarjetas gráficas, de forma que si los algoritmos son escalables se ejecutarán con mayor rapidez en un futuro cercano. En la Figura 2.1 se puede diferenciar el número de procesadores/cores de dos arquitecturas tipo CPU y GPU. Parece razonable pensar que la concepción que tenemos hoy en día de la computación de altas prestaciones irá evolucionando por lo que se hace imprescindible abordar y explorar los beneficios computacionales de las GPU en ámbitos primordiales para el desarrollo humano como la seguridad aérea ante la previsible necesidad de movilidad creciente.

Figura 2.1 Procesadores en las diferentes unidades de proceso



Las principales diferencias entre las CPUs y las GPUs aparecen en el uso operativo de las capacidades tecnológicas (uso del transistor). Las CPUs actuales dedican gran parte de los recursos al control de flujo (predicción dinámica de saltos, reordenamiento dinámico de instrucciones, ejecución especulativa, etc.) y jerarquía de memoria para datos.

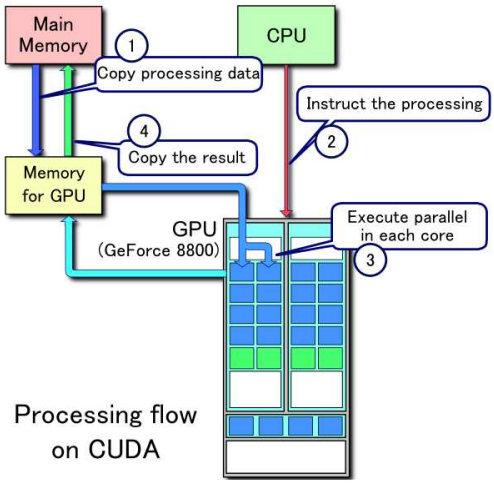
Las GPUs están especializadas en tareas de cómputo intensivo y paralelo, por lo que son óptimas para aplicaciones que trabajen con enormes cantidades de datos de entrada y salida, dónde la misma sección de código se pueda ejecutar a la vez sobre un gran número de elementos de datos (el orden de los cálculos no debe afectar al resultado), con alta densidad aritmética y con una pequeña dependencia entre diferentes elementos de los datos.

El desarrollo de programas en GPU solía ser criticado debido fundamentalmente a la dificultad de los conceptos de bajo nivel que eran necesarios para entender la arquitectura. Debido a este problema, los esfuerzos de parte de la comunidad investigadora e industrial se centraron en crear una arquitectura de cálculo paralelo. Fue NVIDIA entre los años 2006 y 2007 quien liberó CUDA para facilitar el desarrollo de aplicaciones GPGPU. Un año más tarde, Apple propuso la especificación original de OpenCL, desarrollada en conjunto con AMD, IBM, Intel y NVIDIA y en la actualidad dicha propuesta puede ser considerado como un estándar abierto y libre de derechos distribuida por el Grupo Khronos. OpenCL es un framework capaz de ejecutar programas a través de plataformas heterogéneas constituidas por CPUs, GPUs y otros tipos de procesadores.

2.2 - Modelo de CUDA

CUDA (Compute Unified Device Arquitecture) es una arquitectura software y hardware creada para afrontar y administrar los cómputos en la GPU como un dispositivo de cómputo de datos en paralelo, sin necesidad de tener que mapearlos por medio de una API gráfica.

Figura 2.2 Flujo de procesamiento en CUDA



Modelo de programación

La API de CUDA [18] se corresponde con una extensión del lenguaje C, para facilitar su aprendizaje e incorporación a otros proyectos, aunque también existen multitud de wrappers para usar muchos otros lenguajes de programación.

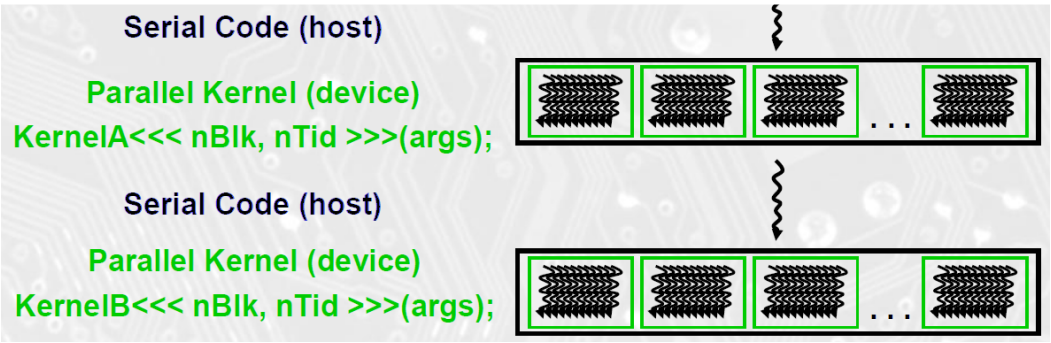
Como muestra la Figura 2.2 anterior, el programa CUDA comienza en el procesador CPU (host). Se invoca a funciones definidas en el API para hacer la reserva (*cudaMalloc*) y transferencia de datos necesarios de entrada entre las memoria principal y la memoria de la GPU (*cudaMemcpy* con el parámetro *cudaMemcpyHostToDevice* en este caso).

Una vez se cuenta con los datos disponibles en la GPU (device) se ejecuta un kernel, que es la función que procesará de manera paralela mediante hilos los datos de entrada. Al terminar, la CPU recupera el control del programa, desde dónde se hace la transferencia del resultado entre las memorias de nuevo mediante *cudaMemcpy* (en este caso, con el parámetro

cudaMemcpyDeviceToHost). Finalmente, se liberan de memoria los datos antes de terminar el programa (*cudaFree*).

El proceso de paralelización en la GPU se puede dividir en distintas etapas que ejecuten kernel iguales o diferentes para distintos datos de entrada. Es la interacción entre el host y el device una de las partes más importantes a definir como se intuye en la Figura 2.3:

Figura 2.3 Ejecución secuencial y paralela entre host y device



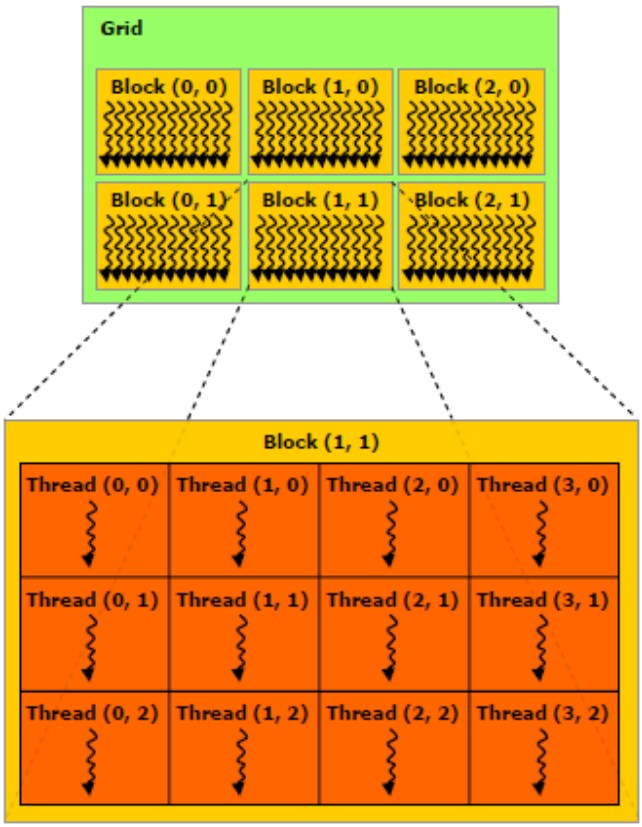
Un kernel se ejecuta a través de un conjunto de hilos paralelos, que son organizados en grids de bloques. En un bloque, los hilos pueden cooperar juntos compartiendo datos eficientemente entre ellos a través de una memoria compartida muy rápida y sincronizando su ejecución para coordinar dichos accesos a memoria. Al invocar un kernel tenemos que especificar sus dimensiones mediante dos valores, DimGrid y DimBlock, asegurándonos que hay suficientes hilos que cubran todos los elementos. El kernel se define como *__global__* puesto que será invocado desde el host para ser ejecutado en el device. Es el prefijo de la declaración, como muestra la Tabla 2.1, el que diferencia a cada función:

Tabla 2.1 Declaración de funciones en CUDA

	Ejecutada en:	**Invocada desde:**
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

La jerarquía de hilos se puede definir mediante valores de DimGrid y DimBlock de 1D, 2D y 3D. Este concepto es interesante, pues en cada hilo dispondremos de una serie de índices para acceder a los datos concretos como threadIdx o blockIdx, como se muestra en la Figura 2.4:

Figura 2.4 Diagrama de jerarquía de hilos y e índices de hilo y bloque



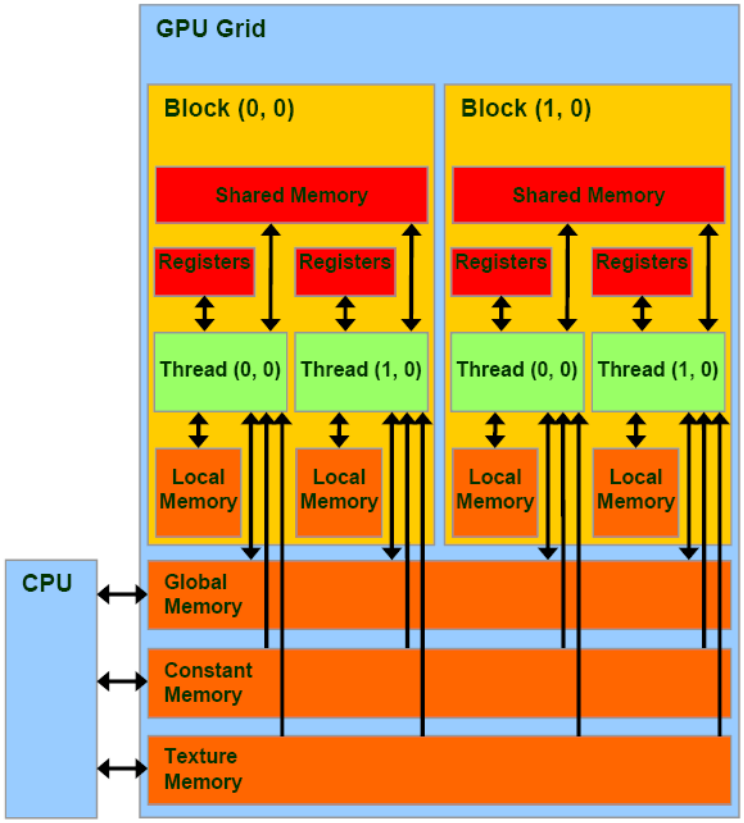

```
int i = threadIdx.x;

int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Modelo de memoria

Cada hilo tiene acceso a una memoria DRAM local privada, y cada bloque tiene acceso a la memoria compartida de la que se ha hablado anteriormente. Además, todos los hilos tienen acceso también a una memoria global. Existen también otros espacios de memoria: la memoria constante y la memoria de texturas.

Entender la jerarquía de memoria es uno de los detalles más interesantes para poder aplicar buenas optimizaciones del código paralelo. La figura 2.5 muestra el modelo de memoria y a continuación se describirán sus características:

Figura 2.5 Modelo de memoria



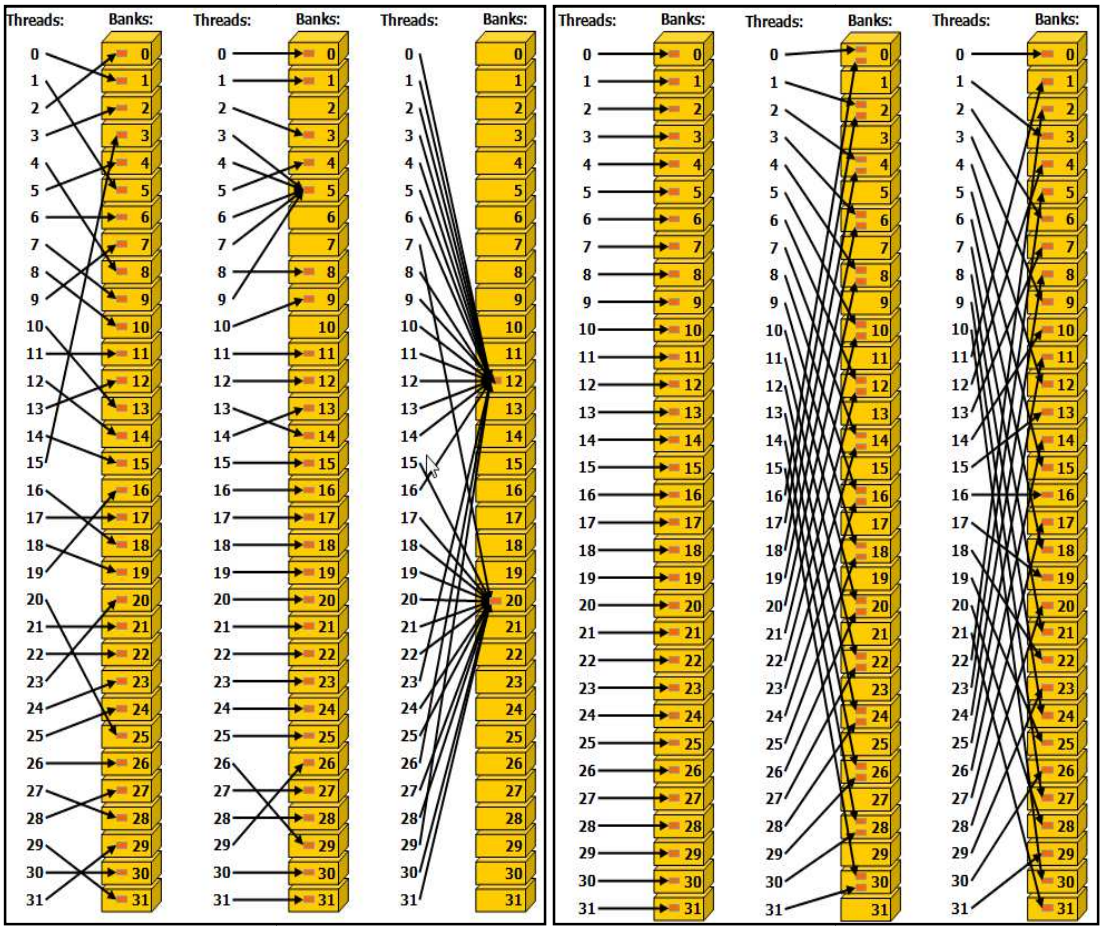
La memoria global es bastante más lenta que el resto, debido principalmente a su mayor tamaño. Se encuentra fuera del chip, al igual que el espacio de memoria de texturas y el espacio de memoria de constantes. La memoria global es accesible para lectura y escritura, y el acceso a sus datos viene determinado por cada tipo de arquitectura en la tarjeta gráfica.

La memoria local también tiene una latencia importante. Es utilizada automáticamente por el compilador para alojar variables que no caben en registros.

Tanto el espacio de memoria constante como el de texturas se encuentran fuera del chip, y, sin embargo, están cacheados. El acceso puede ser costoso como en memoria global, aunque sólo si se produce un fallo de cache.

Por último, la memoria compartida está dentro del chip, una por Streaming Multiprocessor, con accesos mucho más rápidos. Se declara directamente en el código mediante el prefijo __*shared*__. Es importante tratar de minimizar en la medida de lo posible accesos que provoquen conflictos. En la Figura 2.6 se observan ejemplos de accesos a memoria con y sin conflictos:

Figura 2.6 Accesos a memoria compartida



Modelo de ejecución

El planificador de hilos distribuye los bloques de hilos (llamados *warps*) entre los Streaming Multiprocessors (SM), asignando los bloques y usando rodajas de tiempo. Este manejador tiene la responsabilidad de crearlos, distribuirlos y ejecutarlos de forma uniforme y es modificado periódicamente de forma automática para maximizar el uso de los multiprocesadores.

2.3 - Optimización

El objetivo principal de usar GPUs para la resolución del problema de la comparación de trayectorias es mejorar el tiempo de ejecución del algoritmo. Una simple codificación paralela de este algoritmo puede mejorar su implementación secuencial en una CPU, pero este trabajo trata de aprovechar las características especiales de la GPU para incrementar esta mejora.

La escalabilidad es un factor a tener muy en cuenta. Todas las optimizaciones deben tener en cuenta que:

- El mismo algoritmo deberá ser más eficiente sobre una nueva generación de tarjetas gráficas.
- El mismo algoritmo deberá ser más eficiente si se ejecuta en un número mayor de núcleos del mismo tipo.

El rendimiento crecerá en próximas generaciones debido a:

- Incremento del número de cores
- Incremento del número de threads ejecutados en paralelo
- Incremento de la profundidad del pipeline
- Incremento del tamaño de la memoria DRAM
- Incremento del número de canales de memoria DRAM
- Decremento de la latencia de la transferencia de datos

Las recomendaciones que se han seguido [15] para optimizar el problema son las siguientes:

1. Dividir el problema secuencial en partes que pueden procesarse de forma paralela y mantener en el host (CPU) las partes que no son candidatas a paralelizar (Capítulos 3 y 4), Comparación de Trayectorias y Paralelismo, respectivamente.

2. Reducir al mínimo la transferencia de datos entre el host y el dispositivo (Capítulo 4). Paralelismo en los algoritmos de Comparación de Trayectorias.
3. Ajustar la configuración de inicio del kernel para maximizar el uso del dispositivo (Capítulo 5).
4. Optimizar el acceso a memoria, eligiendo la apropiada en cada caso y minimizar los accesos a la memoria global siempre que sea posible (Capítulo 5).
5. Evitar los diferentes caminos de ejecución dentro de un mismo warp y usar patrones de programación paralela y primitivas cuando resulte adecuado (Capítulo 5).

2.4 - Estudio de Hardware

Tras un amplio estudio de las posibilidades disponibles, en este trabajo será usada una Tarjeta gráfica NVIDIA GeForce GTX 660. Ha sido montada en un PC junto a los siguientes componentes principales:

- Procesador Intel Core i5 3330 3.0 Ghz con 6MB de cache y 4 núcleos.
- Memoria principal de 8GB DDR3 Kingston a 1600 Mhz.

El sistema operativo utilizado ha sido CentOS 6.4, basado en la distribución Red Hat Enterprise Linux RHEL. El entorno de programación está compuesto del editor Emacs y el compilador NVIDIA CUDA Compiler (nvcc) release 5.0, V0.2.1221.

Los drivers utilizados para la tarjeta gráfica son los oficiales de NVIDIA para este sistema operativo, versión 310.40. La arquitectura de la GPU es Kepler, una arquitectura de computación de alto rendimiento y pensada para su uso en aplicaciones científicas.

Cuenta con 960 núcleos CUDA, cada uno de los cuales incorpora una unidad aritmético lógica (ALU) y una unidad de cálculo en coma flotante (FPU). Permite usar aritmética de doble precisión, algo que no se soportaba en los inicios de CUDA y que es un requisito imprescincible para nuestro problema. Tiene una frecuencia de reloj de 980 MHz. Un característica importante de esta tarjeta es que detecta automáticamente la carga y aumenta la frecuencia del reloj cuando es posible, hasta 1033MHz.

Otra característica interesante de esta tarjeta es su memoria: 2GB de memoria, con una frecuencia de 6.0 Gbps. La interfaz es de 192-bit GDDR5 y el ancho de banda máximo 144.2GB/s. Como veremos, además de la capacidad de cálculo, la tasa de transferencia de memoria y su cantidad, así como su disposición, juega un papel importantísimo.

Kepler es una arquitectura modular con la estructura que se observa en la Figura 2.7, y cuya base son los Next Generation Streaming Multiprocessors (SMX), los que están conformados por un total de 192 shader processors cada uno, cifra muy superior a los 32 o 48 de los Streaming Multiprocessors (SM) de la anterior arquitectura Fermi. Además, cada SMX posee una unidad PolyMorphEngine 2.0, la cual se encarga de procesar la geometría y teselado, ofreciendo un rendimiento que duplica al de las unidades PolyMorphEngine de Fermi. La figura 2.8 muestra la estructura completa. Por último y no por ello menos importante, cada SMX posee un total de 16 nuevas unidades de textura las que poseen un modelo bindless, pudiendo acceder a más de 128 texturas simultáneas, y realizando un menor uso de CPU.

Figura 2.7 Diagrama del chip de la Arquitectura Kepler

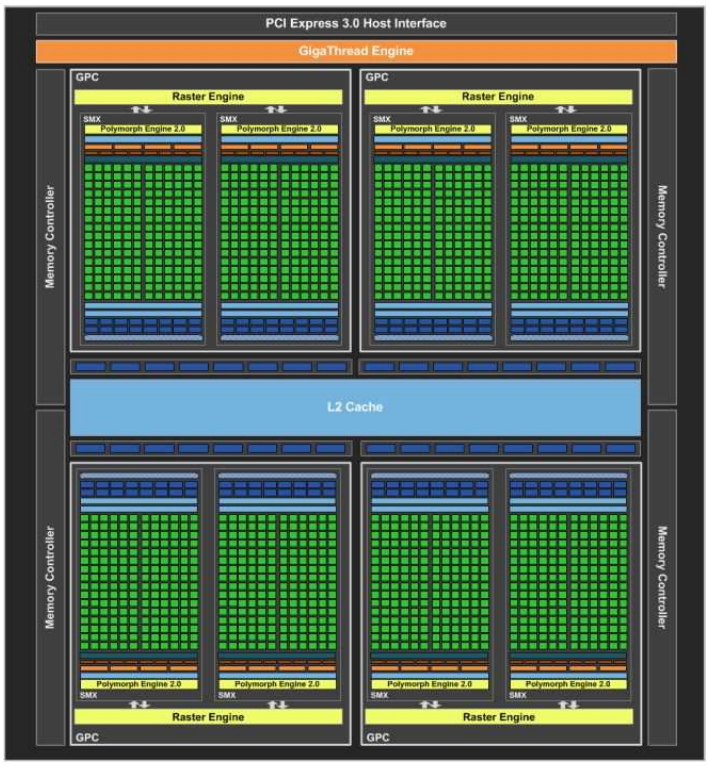
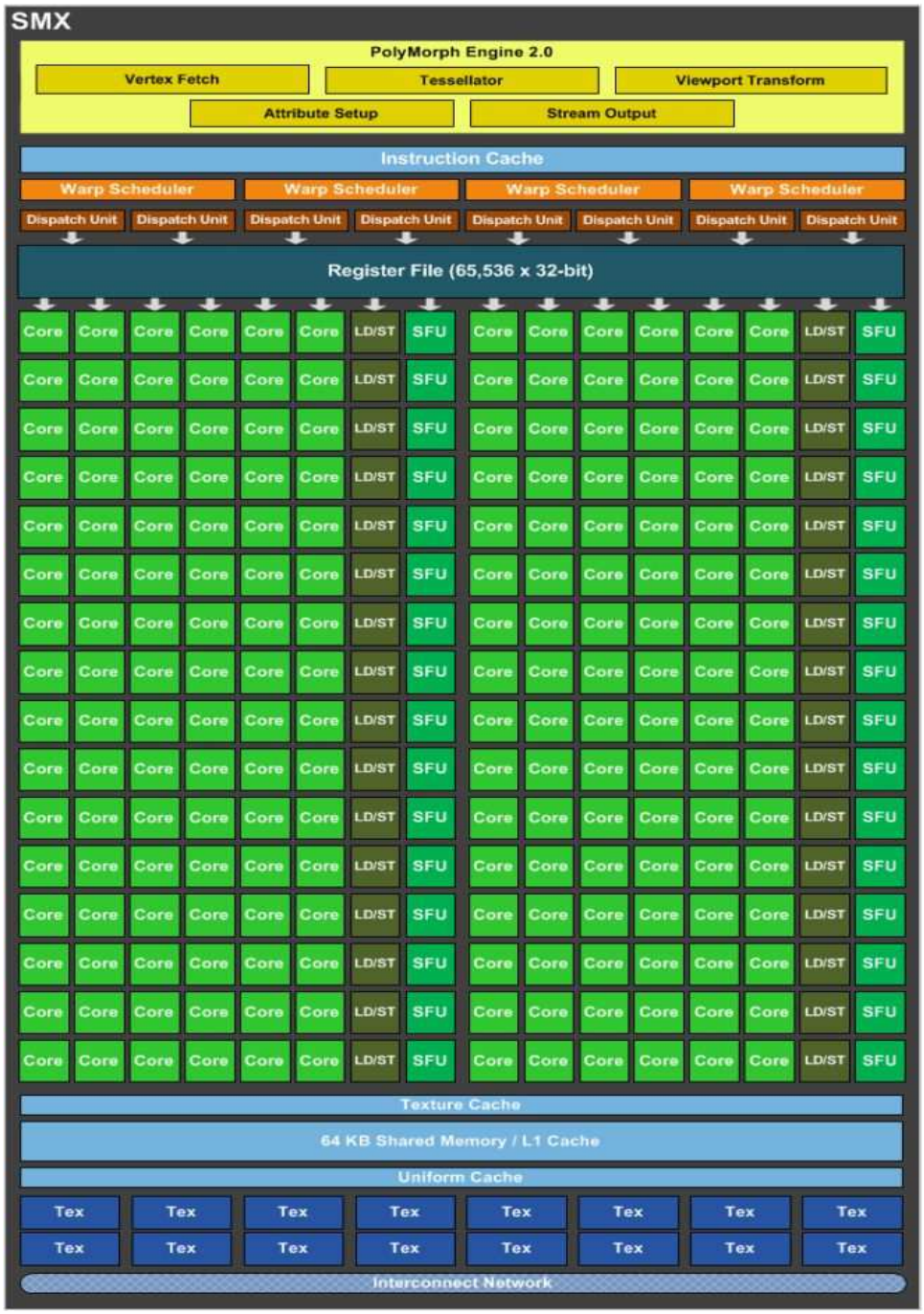


Figura 2.8 Streaming Multiprocessor



3 - Comparación de Trayectorias

Con el objetivo de acelerar el algoritmo general de comparación de trayectorias se propone una metodología basada en un conjunto de etapas paralelizables. Para abordar esta división del problema se introducen algunos conceptos clave que permitan el entendimiento de las decisiones tomadas en capítulos posteriores.

3.1 -Extended Projected Profile (EPP)

La Aplicación ADS (Automatic Dependent Surveillance) es el mecanismo por el cual la aeronave envía su trayectoria al sistema de tierra periódicamente, por demanda o ante eventos destacables mediante un ADS-C Report que contiene un EPP (Extended Projected Profile). Este EPP es calculado automáticamente por el FMS (Flight Management System) y contiene una lista completa de puntos desde la posición actual hasta el umbral de la pista de aterrizaje en el aeropuerto de destino (entre 1 y 128 puntos según el estándar AGDL). Estos son los puntos destacables para el vuelo, no sólo los pertenecientes al plan de vuelo, sino aquellos interesantes por diversos motivos (cambios de rumbo, cambios de altitud, entradas y salidas de aerovías, etc.).

Para cada uno de estos puntos, se provee la siguiente información:

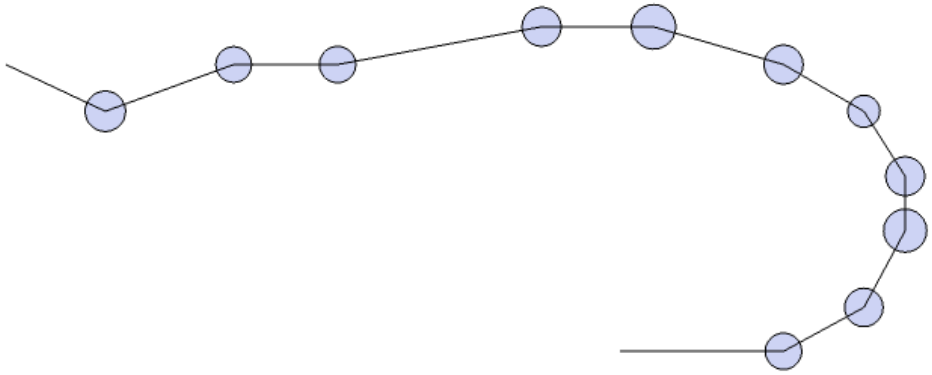
- Posición (latitud y longitud)
- Nivel (altitud)
- Tiempo estimado de sobrevuelo
- Tipo de punto (por ejemplo Top of Climb, Top of Descent)

Además, el EPP puede contener otro tipo de información, como la velocidad estimada en cada punto, información meteorológica, masa actual de la aeronave, etc..

Por medio del EPP se puede trazar la trayectoria futura completa de la aeronave, como muestra el ejemplo de la Figura 3.1. En caso de cualquier mandato por parte del controlador aéreo que modifique esta ruta debido a muchas posibles causas (probabilidad de conflicto con otra aeronave, previsión meteorológica, congestión del tráfico) el propio FMS computará los

cambios automáticamente y proveerá de un nuevo EPP con el que actualizar su nueva trayectoria.

Figura 3.1 Extended Projected Profile de 11 puntos sobre la trayectoria



3.2 - Sampling

Hasta ahora se ha hablado teniendo en cuenta la trayectoria de una sola aeronave. Para calcular el conflicto entre dos aeronaves se va a necesitar una forma de comparar dos EPP, que a priori son bastante diferentes, porque cada uno contiene diferentes puntos en diferentes tiempos. Se podría plantear una solución geométrica secuencial para resolver esta cuestión. Ésta es, por norma general, la manera comúnmente más usada [14]:

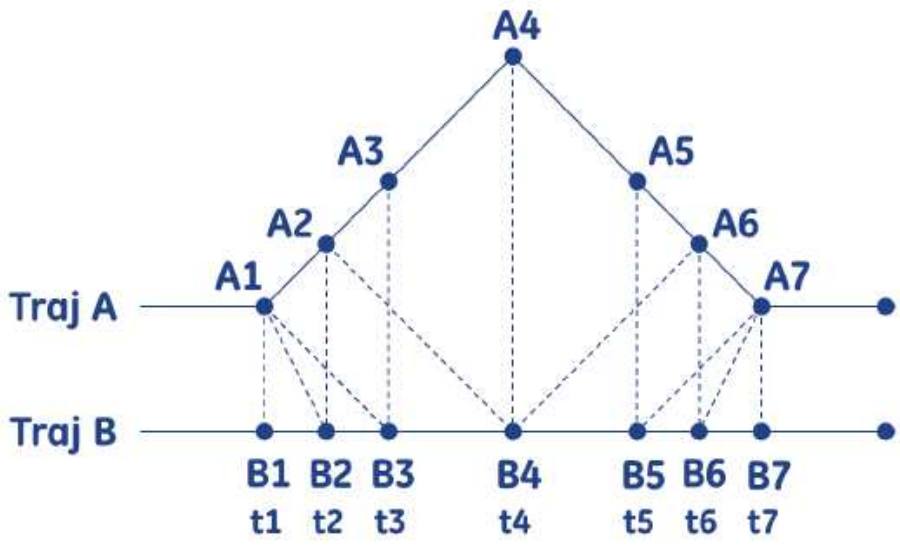
Uniendo los puntos adyacentes de un EPP mediante segmentos se podrían calcular todas las combinaciones de segmentos entre las dos trayectorias y comprobar si existen puntos de corte. Si existen puntos de corte en un intervalo de tiempo común para ambas trayectorias es cuando se puede hablar de conflicto. Se considera un conflicto entre aeronaves cuando sus trayectorias se encuentran lo suficientemente próximas no respetando el mínimo de separación que se considera segura.

Las aeronaves deben de contar con un espacio de seguridad vertical y horizontal que no debe ser violado por otras aeronaves cercanas. El problema reducido a la comparación de trayectorias de todos los posibles segmentos conlleva un coste muy elevado motivado por el alto número de posibles puntos de intersección. Si se simplifica este problema a 2D, los segmentos

entre puntos adyacentes del EPP se convierten en rectángulos planos con los que calcular puntos de corte. Si tenemos en cuenta el problema en 3D, estos rectángulos se suelen convertir por norma general en cilindros que mantienen la distancia de separación vertical y horizontal entre aeronaves, aumentando de forma considerable el coste de los cálculos asociados para encontrar puntos de corte.

En este caso, la intención de este trabajo para paralelizar la comparación de trayectorias está basada en utilizar primero un algoritmo paralelo de muestreo, por el que transformar el EPP en una serie de puntos que puedan ser temporalmente coincidentes con los de las trayectorias, como podemos observar a continuación en la Figura 3.2:

Figura 3.2 Muestreo de EPP aplicado a dos trayectorias



La "sincronización" de trayectorias en esta figura consiste en calcular la posición de ambas aeronaves en tiempos coincidentes t_1, t_2,..t_n para poder calcular, por ejemplo, la distancia que separa a dichas aeronaves en un determinado instante. Esta “sincronización” entre trayectorias puede hacerse por interpolación. Se usarán diversos métodos de interpolación con este objetivo. Como los puntos introducidos por el FMS en el EPP son los más significativos de

la trayectoria habrá que estudiar posibles formas de obtener mejores resultados para después hacer una comparación geométrica o analítica entre varias aeronaves.

Las interpolaciones candidatas a utilizar son las siguientes:

- **Interpolación lineal** con gran tasa de muestreo
- **Interpolación cuadrática**
- **Interpolación polinómica o Spline** con menor tasa de muestreo

Se relacionará cada una de las interpolaciones con la distancia de seguridad admisible para cada aeronave teniendo en cuenta la tasa de muestreo que el propio método de interpolación recomiende en cada caso.

De forma estandarizada, los datos de posición de cada punto obtenidos en el EPP tienen el formato de latitud y longitud y altitud. Para aplicar los métodos de interpolación con respecto al tiempo para cada una de las variables anteriores se debe hacer un cambio de coordenadas geodésicas a cartesianas. Este algoritmo podría paralelizarse también debido a que cada cambio de coordenadas es independiente a cada punto de la trayectoria.

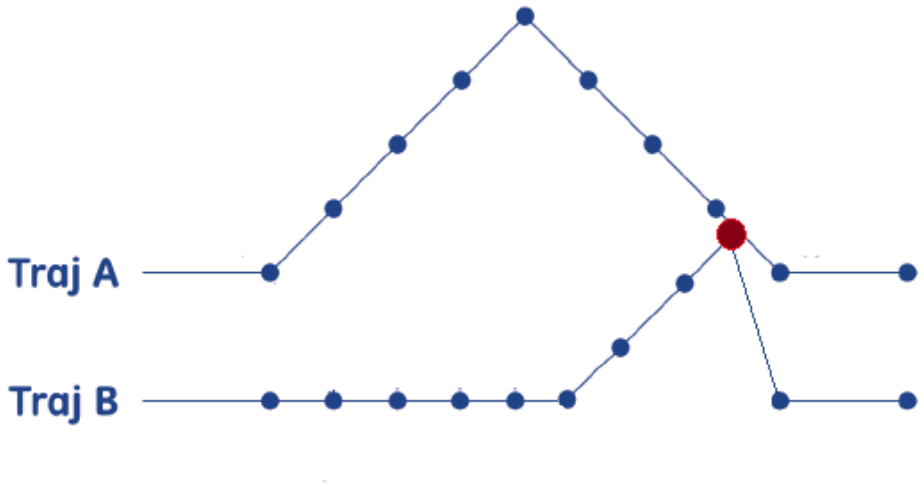
3.3 - Geometría simplificada de conflicto

Una vez se tengan dos trayectorias muestreadas de dos EPP de diferentes aeronaves habrá que procesarlas buscando conflictos. En un instante t, podemos calcular si una aeronave se encontrará posicionada dentro del espacio de seguridad de la otra aeronave de forma geométrica. Esta es la operación básica a ejecutar en cada procesador de GPU de forma paralela. Además habrá que tener en cuenta otras restricciones dependiendo del algoritmo de muestreo utilizado.

Si la tasa de muestreo utilizada anteriormente es alta, con esta operación básica debería ser suficiente para asegurar que encontramos todos los posibles conflictos. Si la tasa de muestreo es menor tendremos que tener en cuenta puntos adyacentes para verificar que los instantes previos o posteriores a t están libres de conflicto también.

Resulta relevante intentar hacer esta comprobación geométrica en 3D, debido a que las trayectorias pueden sufrir tanto discrepancias verticales como horizontales. El objetivo pasa por conseguir la mayor eficiencia el algoritmo encontrando un compromiso entre la tasa de muestreo y la distancia de seguridad. Una alta tasa de muestreo aumenta el número de operaciones a realizar, disminuye la distancia de seguridad entre aeronaves, y simplifica el cálculo principal hasta llegar a una operación atómica idealmente paralelizable como pueda ser calcular la distancia entre dos puntos. Por el contrario, una tasa de muestreo menor se queda a medio camino entre la paralelización absoluta y la necesidad de compartir más información entre los cálculos, o bien obliga a aumentar la distancia de seguridad entre aeronaves:

Figura 3.3 Importancia de la tasa de muestreo en la paralelización



El punto rojo de conflicto marcado en la Figura 3.3 podría haber sido detectado mediante una operación geométrica simple con una alta tasa de muestreo o mediante el aumento de la distancia de seguridad entre las trayectorias A y B.

3.4 - Detección múltiple de conflictos

De un modo general se debería poder usar otro algoritmo paralelo para la detección de conflictos de múltiples aeronaves teniendo en cuenta conceptos del espacio aéreo. El espacio aéreo está dividido de forma dinámica, donde cada porción de espacio es controlada por un controlador aéreo diferente bajo el concepto de AoI (Área de interés) y AoR (Área de responsabilidad). Cabe destacar que detectar conflictos debe ser una capacidad distribuida entre cada uno de los sectores de dicho espacio aéreo, puesto que cada controlador está interesado solamente en los conflictos que puedan acontecer en su área de responsabilidad. Existen algunas aproximaciones como [9] que han estudiado algoritmos distribuidos para la planificación de trayectorias en UAVs (Unmanned Aerial Vehicles, Vehículos Aéreos no Tripulados).

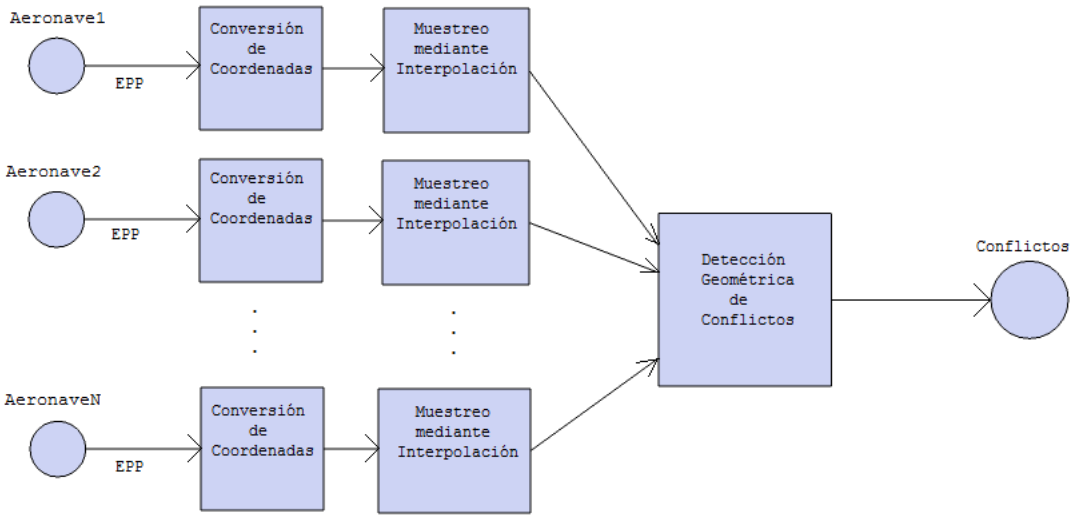
Un sistema integral de tráfico aéreo puede tener requisitos de capacidad de hasta 6000 vuelos concurrentes en un pequeño espacio de tiempo, preparados para dar soporte a los espacios aéreos cuya congestión es mayor. Las aproximaciones de grandes capitales europeas como París, Londres o Bruselas condensan una gran cantidad de vuelos teniendo en cuenta también el número de ciudades que cuentan con aeropuertos en zonas relativamente pequeñas y cercanas a ellas, como muestra la Figura 3.4. Además, muchos vuelos de larga distancia atraviesan estas zonas también.

Figura 3.4 Congestión de tráfico aéreo en Europa

4 - Paralelismo en los algoritmos de Comparación de Trayectorias

El objetivo es comprobar si una división del problema en etapas paralelizables constituye una sólida alternativa a su implementación secuencial. A partir de los conceptos explicados en el capítulo 3, se propone el siguiente esquema:

Figura 4.1 Etapas paralelizables de la Comparación de Trayectorias



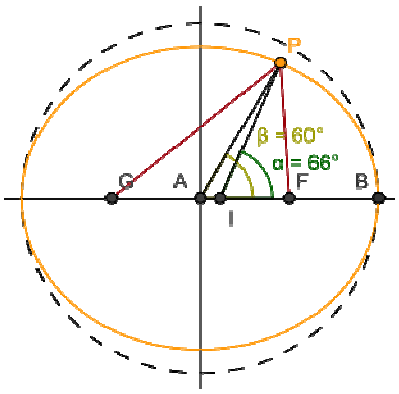
A continuación se explica cada una de las etapas de la Figura 4.1 de forma detallada para una aeronave. La intención es finalizar el capítulo con la comparación de trayectorias global para la obtención de conflictos, tratando de estudiar la escalabilidad del problema conjunto en cuanto a número de aeronaves.

4.1 -Algoritmo de conversión de coordenadas

La entrada de esta etapa es un EPP, conjunto de puntos cuyas coordenadas geodésicas recibidas vienen definidas en latitud, longitud y altitud, comúnmente usadas en navegación. En el caso de las dos primeras, las unidades son grados, minutos y segundos. Es conveniente realizar un cambio de coordenadas a cartesianas (x, y, z) puesto que los cálculos geométricos que describe el problema de la sincronización de trayectorias serán simplificados.

Teniendo en cuenta que la Tierra es un elipsoide imperfecto, se usa el estándar WGS 84 (World Geodetic System) como sistema de referencia para la conversión representada en la Figura 4.2.

Figura 4.2 Conversión de coordenadas



La salida esperada de la conversión es del mismo tamaño que el conjunto de puntos recibidos como entrada, pero siendo cada punto definido mediante sus coordenadas cartesianas en 3D (x, y, z), y manteniendo el resto de información recibida en el EPP.

El algoritmo que se explicará con detalle a continuación se compone de operaciones aritméticas y trigonométricas simples para cada punto, usando los parámetros de diseño que se muestran en la Tabla 4.1 de la siguiente página.

Tabla 4.1 Parámetros definidos por WGS 84

Parameter	**Notation**	**Value**
Semi-major axis	A	6378137.0 m
Reciprocal of flattening	1/f	298.257223563
Semi-minor axis	B	6356752.3142 m
First eccentricity squared	e^2	6.694379990 14*10^{-3}

Fijados los parámetros de diseño de la tabla anterior, las ecuaciones de conversión de coordenadas geodésicas a cartesianas se explican a continuación.

Dada la latitud ϕ y la longitud λ de un punto en grados ϕ_d y λ_d, minutos ϕ_m y λ_m ,y segundos ϕ_s y λ_s y su altura (h):

$$\phi = (\phi_d + \phi_m / 60 + \phi_s / 3600) * \pi / 180 \qquad (1)$$
$$\lambda = (\lambda_d + \lambda_m / 60 + \lambda_s / 3600) * \pi / 180 \qquad (2)$$

las coordenadas cartesianas equivalentes son:

$$\begin{aligned} x &= (N(\phi) + h)\cos\phi\cos\lambda \\ y &= (N(\phi) + h)\cos\phi\sin\lambda \\ z &= (N(\phi)(1-e^2) + h)\sin\phi \end{aligned} \qquad (3)$$

donde

$$N(\phi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}} \qquad (4)$$

La conversión de coordenadas cuenta con buenas condiciones para ser paralelizable en cuanto a las operaciones a realizar: No tiene dependencia de datos entre ellas y no hay reuso de datos.

Sin embargo, el paralelismo de datos debe ser tenido muy en cuenta en este caso, porque el número de puntos de entrada es bastante reducido.

4.2 - Algoritmos de interpolación

Como segunda etapa, la intención es obtener un gran número de puntos que describan la trayectoria de la aeronave con respecto al tiempo a partir del EPP compuesto de puntos con coordenadas cartesianas que se calcula en la etapa anterior. Esto se puede conseguir por medio de diferentes métodos de interpolación:

4.2.1 - Interpolación lineal

La interpolación es un método sencillo y viable para obtener puntos intermedios de la trayectoria de una aeronave. Dados dos puntos conocidos de la trayectoria mediante un EPP recibido, la línea recta que los une aproxima los puntos intermedios que se quieren obtener.

Se quiere obtener la interpolación de la posición de una aeronave en un tiempo t para x, y, z. Puesto que los 128 puntos recibidos en el EPP son por definición los puntos más destacables de la trayectoria (cambios de rumbo, de nivel, de velocidad, top of descent, top of climb), aún usando la sencilla interpolación lineal vamos a evitar grandes errores en la predicción de trayectoria.

En este caso, la implementación de este algoritmo en GPU es eficiente. Para cada elemento, dados dos puntos de trayectoria (x_{i-1}, y_{i-1}, z_{i-1}) y (x_i, y_i, z_i) por los que la aeronave pasa en t_{i-1} y t_i respectivamente, podemos obtener un punto interpolado (x, y, z) en t, siendo $t_{i-1}<t<t_i$ mediante las ecuaciones:

$$x = x_{i-1} + (t - t_{i-1})\frac{x_i - x_{i-1}}{t_i - t_{i-1}} \qquad y = y_{i-1} + (t - t_{i-1})\frac{y_i - y_{i-1}}{t_i - t_{i-1}} \qquad z = z_{i-1} + (t - t_{i-1})\frac{z_i - z_{i-1}}{t_i - t_{i-1}} \quad (5)$$

Como los intervalos de tiempo para los que se quiere hacer la comparación de trayectorias deben ser muy reducidos, se van a obtener una gran cantidad de puntos a partir del EPP. Ese número de puntos obtenidos es el número de operaciones que tendremos que hacer (para cada coordenada), y si es suficientemente alto, el rendimiento de efectuar estas operaciones en GPU debería ser superior a efectuarlas mediante CPU, cosa que veremos en el Capítulo 5.2.

4.2.2 - Interpolación cuadrática

A pesar de la buena aproximación que supone hacer una simple interpolación lineal, se puede mejorar el muestreo con respecto al verdadero movimiento de la aeronave incluyendo otras variables como la velocidad. El EPP ofrece la velocidad estimada de la aeronave en cada punto de trayectoria, por lo que se puede mejorar el algoritmo anterior mediante un método de interpolación de 2º orden en el que se asume una aceleración constante. En caso de que la velocidad no sea disponible en alguno de los puntos del EPP, este método degenera en una interpolación lineal.

Para cada elemento, dados dos puntos de trayectoria (x_{i-1}, y_{i-1}, z_{i-1}) y (x_i, y_i, z_i) por los que la aeronave pasa en t_{i-1} y t_i a velocidades v_{i-1} y v_i respectivamente, podemos obtener un punto interpolado (x, y, z) en t, siendo $t_{i-1}<t<t_i$ mediante las ecuaciones:

$$x = \frac{Ax_{i-1} + Bx_i}{C} \qquad y = \frac{Dy_{i-1} + Ey_i}{F} \qquad z = z_{i-1} + (t - t_{i-1})\frac{z_i - z_{i-1}}{t_i - t_{i-1}} \tag{6}$$

siendo:

$$A = (v_{i-1(x)} - v_{i(x)})(t_i - t)^2 + 2v_{i(x)}(t_i - t_{i-1})(t_i - t) \tag{7}$$

$$B = (v_{i(x)} - v_{i-1(x)})(t - t_{i-1})^2 + 2v_{i-1(x)}(t_i - t_{i-1})(t - t_{i-1}) \tag{8}$$

$$C = (v_{i-1(x)} + v_{i(x)})(t_i - t_{i-1})^2 \tag{9}$$

$$D = (v_{i-1(y)} - v_{i(y)})(t_i - t)^2 + 2v_{i(y)}(t_i - t_{i-1})(t_i - t) \tag{10}$$

$$E = (v_{i(y)} - v_{i-1(y)})(t - t_{i-1})^2 + 2v_{i-1(y)}(t_i - t_{i-1})(t - t_{i-1}) \tag{11}$$

$$F = (v_{i-1(y)} + v_{i(y)})(t_i - t_{i-1})^2 \tag{12}$$

Para z basta una interpolación lineal para simplificar el cálculo de coeficientes. La tasa de muestreo puede ser la misma que para la interpolación lineal, pero como el algoritmo a realizar para cada punto es un poco más costoso, deberíamos obtener mejor speedup con respecto a la CPU que en el caso anterior, aunque aumentando el tiempo de cómputo puesto que el muestreo es bastante más realista. El tiempo de cómputo aumenta porque resulta necesario hacer mayor número de cálculos para obtener los coeficientes (A, B, C, D, E, F). Por otra parte, el muestreo es más realista debido a que se usa la velocidad para interpolar los puntos con mayor exactitud.

4.2.3 - Interpolación polinómica

Para representar curvas es común el uso de polinomios por intervalos, cuya función es llamada spline. Esta técnica es usada habitualmente en informática y ha sido utilizada en el mundo aeronáutico por suavizar las trayectorias. Se usará la versión de splines cúbicas debido a que se obtienen resultados más adecuados a las curvas mediante polinomios de grado bajo y se evitan indeseables oscilaciones.

La idea central es que en vez de usar un solo polinomio para interpolar los datos, se pueden usar segmentos de polinomios y unirlos adecuadamente (bajo ciertas condiciones de continuidad) para formar la interpolación. La función spline cúbica es una función s(t) definida así:

$$s(t) = \begin{cases} s_0(t) & si \quad t \in [t_0, t_1] \\ s_1(t) & si \quad t \in [t_1, t_2] \\ \vdots & \\ s_{n-1}(t) & si \quad t \in [t_{n-1}, t_n] \end{cases}$$

donde cada $s_i(t)$ es un polinomio cúbico que cumple $s_i(t_i) = x_i$, para todo i = 0,1,..,n y tal que s(t) tiene primera y segunda derivadas continuas en $[t_0,t_n]$. En nuestro problema de 3D, necesitaremos tres interpolaciones, una para cada coordenada, de forma que usaremos la notación $s_x(t)$, $s_y(t)$ y $s_z(t)$ que para referirnos a ellas.

A continuación se expone el método mediante un ejemplo muy sencillo de interpolación de la coordenada x en t. Tenemos como puntos recibidos x=-1, x=2, x=-7 en t=2, t=3 y t=5 respectivamente.

El polinomio $s_x(t)$ se define por intervalos como sigue:

$a_1t^3 + b_1t^2 + c_1t + d_1$ si $t \in [2,3]$

$a_2t^3 + b_2t^2 + c_2t + d_2$ si $t \in [3,5]$

Y hacemos que se cumpla la condición $s_x(t_i) = x_{i,}$

$s_x(2) = -1$

$$\mathbf{8a_1 + 4b_1 + 2c_1 + d_1 = -1} \quad (13)$$

$s_x(3) = 2$

$$\mathbf{27a_1 + 9b_1 + 3c_1 + d_1 = 2} \quad (14)$$

$$\mathbf{27a_2 + 9b_2 + 3c_2 + d_2 = 2} \quad (15)$$

$s_x(5) = -7$

$$\mathbf{125a_2 + 25b_2 + 5c_2 + d_2 = -7} \quad (16)$$

Primera derivada:

$s_x´(t) = 3a_1t^2 + 2b_1t + c_1$ si $t \in [2,3]$

$s_x´(t) = 3a_2t^2 + 2b_2t + c_2$ si $t \in [3,5]$

Para evitar discontinuidad en t=3, tenemos:

$3a_1(3)^2 + 2b_1(3) + c_1 = 3a_2(3)^2 + 2b_2(3) + c_2$

$$\mathbf{27a_1 + 6b_1 + c_1 = 27a_2 + 6b_2 + c_2} \quad (17)$$

Segunda derivada:

$s_x´´(t) = 6a_1t + 2b_1$ si $t \in [2,3]$

$s_x´´(t) = 6a_2t + 2b_2$ si $t \in [3,5]$

Para evitar la discontinuidad, de forma análoga:

$6a_1(3) + 2b_1 = 6a_2(3) + 2b_2$

$$\mathbf{18a_1 + 2b_1 = 18a_2 + 2b_2} \quad (18)$$

En este punto contamos con 6 ecuaciones y 8 incógnitas, por lo tanto tenemos 2 grados de libertad. En general, se agregan las siguientes 2 condiciones:

$s_x''(t_0) = 0$

$s_x''(t_n) = 0$

Obteniendo:

$s_x''(2) = 0,\ 6a_1(2) + 2b_1 = 0$

$$\mathbf{12a_1 + 2b_1 = 0} \qquad (19)$$

$s_x''(5) = 0,\ 6a_2(5) + 2b_2 = 0$

$$\mathbf{30a_2 + 2b_2 = 0} \qquad (20)$$

Tenemos un sistema de 8 ecuaciones con 8 incógnitas, expresado de forma matricial:

$$\begin{bmatrix} 8 & 4 & 2 & 1 & 0 & 0 & 0 & 0 \\ 27 & 9 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 27 & 9 & 3 & 1 \\ 0 & 0 & 0 & 0 & 125 & 25 & 5 & 1 \\ 27 & 6 & 1 & 0 & -27 & -6 & -1 & 0 \\ 18 & 2 & 0 & 0 & -18 & -2 & 0 & 0 \\ 12 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 30 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ a_2 \\ b_2 \\ c_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 2 \\ -7 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

cuyas soluciones son:

$a_1 = -1.25$ $b_1 = 7.5$ $c_1 = -10.75$ $d_1 = 0.5$

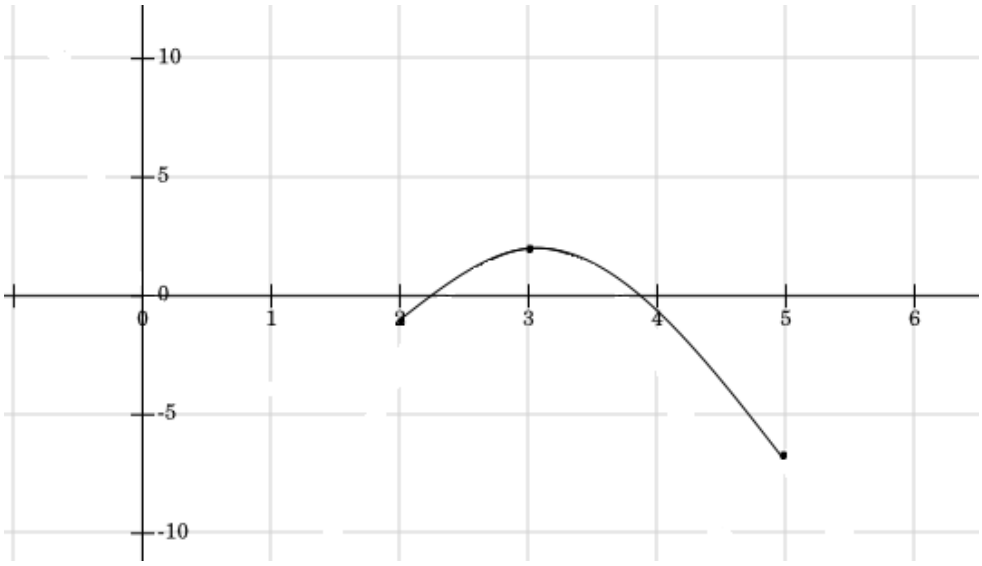
$a_2 = 0.625$ $b_2 = -9.375$ $c_2 = 39.875$ $d_2 = -50.125$

y quedando como spline cúbica:

$$s_x(t) = \begin{cases} -1.25t^3 + 7.5t^2 - 10.75t + 0.5 & si \quad t \in [2,3] \\ 0.625t^3 - 9.375t^2 + 39.875t - 50.125 & si \quad t \in [3,5] \end{cases}$$

Los cálculos son análogos para las interpolaciones $s_y(t)$ y $s_z(t)$.

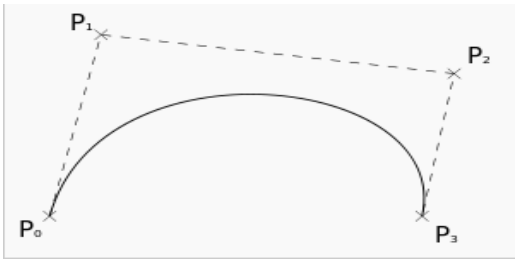
Figura 4.3 Representación gráfica correspondiente a la interpolación $s_x(t)$ desarrollada



Como se puede observar en la Figura 4.3, los polinomios enlazan perfectamente debido a las condiciones impuestas sobre las derivadas.

Para la implementación de esta interpolación se ha decidido hacer uso de un método aproximado de splines cúbicas conocido como Curvas Cúbicas de Bézier, que han sido las más usadas generalmente para este propósito [19]. Para calcular la curva entre dos puntos, P_0 y P_3, se cuenta con dos puntos de control intermedios (puntos de anclaje) P_1 y P_2 como representa la Figura 4.4:

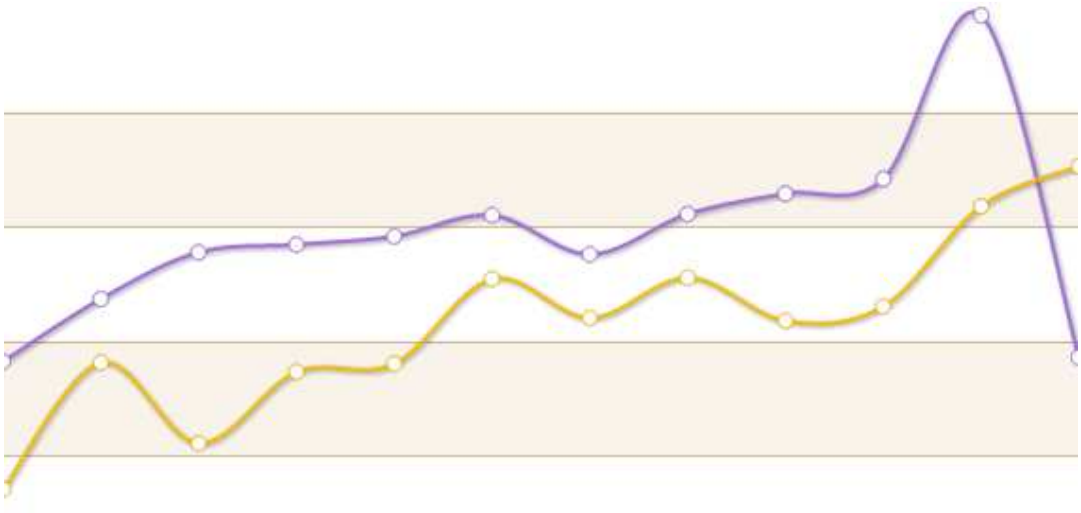
Figura 4.4 Curva de Bézier entre dos puntos



La clave de la paralelización de este método es que además de que la complejidad del cálculo aumenta con respecto a los anteriores, el uso de memoria para obtener la lista de puntos de control (P1 y P2) para cada intervalo incrementa de forma significativa. Si fuera necesario obtener una aproximación de la curva tan buena como la que proporciona este método, el uso coherente de la memoria en la GPU sería imprescindible para alcanzar mejores rendimientos de los que consigue el algoritmo secuencial en la CPU.

Dado que los 128 puntos como máximo que recibimos en el EPP estándar y que tendremos como entrada de la interpolación son por definición los puntos más significativos de la trayectoria, no se espera que sea necesario el uso de este método en una implementación final (teóricamente) puesto que no esperamos grandes oscilaciones en las funciones a muestrear, aunque tiene mucha utilidad su estudio puesto que las condiciones de los puntos que recibimos en el EPP podrían ser diversas y la interpolación conseguida por este método parece que asegura mejores resultados. En la Figura 4.5 se ejemplifica el muestreo esperado al usar esta técnica.

Figura 4.5 Ejemplo de muestreo de trayectorias por interpolación cúbica



4.3 - Algoritmo de detección de conflictos

4.3.1 - Detección geométrica

Para detectar los conflictos entre dos aeronaves, dadas sus trayectorias mediante EPPs, se usa cualquiera de los métodos de muestreo vistos en el apartado anterior. Gracias a este muestreo se puede detectar un posible conflicto comparando punto a punto ambas trayectorias.

Se debe encontrar una relación entre la tasa de muestreo (número de puntos), la calidad de los puntos, y la distancia de seguridad entre aeronaves. Las dos primeras vienen determinadas por el algoritmo de interpolación y la última se puede considerar como un parámetro externo como dato incorporado en el EPP.

- Si la tasa de muestreo es baja, esto es, los puntos de una misma trayectoria están bastante separados, hay dos opciones: aumentar la distancia de separación entre aeronaves o utilizar métodos suplementarios que permitan verificar que ningún conflicto es pasado por alto.

- Si la tasa de muestreo es alta, se puede comparar punto a punto con una distancia de separación mucho menor sin necesidad de añadir otros métodos. Esto permitiría un mejor aprovechamiento del espacio aéreo y, como se intuye teóricamente, cuanta mayor tasa de muestreo obtendremos mejor rendimiento con respecto a un algoritmo secuencial ejecutado en la CPU.

La paralelización en este último caso es simple. Basta con calcular de forma geométrica si la distancia entre dos puntos es mayor o menor que una distancia de separación entre aeronaves definida (el muestreo permite poder calcular de forma eficiente la distancia entre dos puntos tanto en 2D como en 3D, cosa que otros métodos tratan de evitar debido a la complejidad geométrica 3D).

La ecuación de distancia entre 2 puntos $P_1(x_1,y_1,z_1)$ y $P_2(x_2,y_2,z_2)$ según la métrica euclídea es:

$$|P_1P_2| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \qquad (21)$$

Debido a que la distancia de seguridad entre aeronaves puede cambiar dinámicamente, se establece un umbral μ como entrada externa del sistema tal que:

Hay conflicto entre dos aeronaves A, B en t_i cuando $|P_A(t_i)P_B(t_i)| < \mu$.

Se podría optar por considerar μ como una entrada externa del sistema que tiene en cuenta la estela turbulenta de la aeronave, la meteorología y otras condiciones para la detección de conflictos. En este caso lo vamos a usar como parámetro dependiente de un dato que se recibe en el EPP y que contiene la precisión de los puntos calculados por la aeronave, la *Figure of Merit*, medida que caracteriza de forma cuantitativa la utilidad dichos puntos estableciendo su exactitud y cuyos valores se muestran en la Tabla 4.2. La relación entre μ y la *Figure of Merit* (FoM) se estudiará con mayor detalle en el capítulo de resultados.

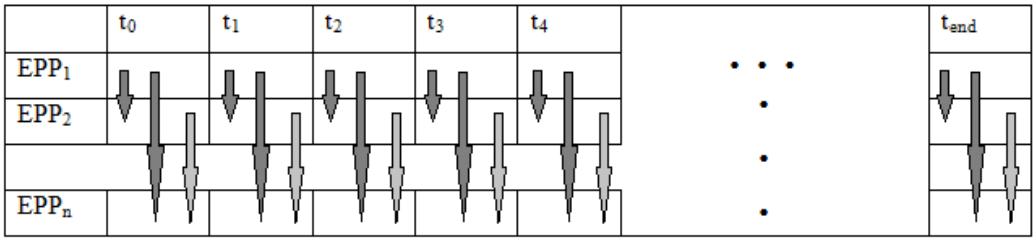
Tabla 4.2 Figure of Merit

Figure of Merit Level	**Precisión de la posición (con un 95% de Probabilidad)**
0	Complete loss
1	< 30 nautical miles
2	< 15 nautical miles
3	< 8 nautical miles
4	< 4 nautical miles
5	< 1 nautical miles
6	< 0.25 nautical miles
7	< 0.05 nautical miles

4.3.2 - Detección Grid-based

Lo más interesante de poder detectar los conflictos usando las técnicas propuestas anteriormente es que se puede tratar de generalizar el problema a N aeronaves y seguir aprovechando la capacidad de paralelización que nos permite la GPU. Por medio de la construcción de un grid espacio-tiempo 4D como el de la Figura 4.6 se pueden computar qué celdas están ocupadas y hacer el chequeo oportuno para detectar conflictos.

Figura 4.6 Grid espacio - tiempo



La primera intención es aplicar una optimización usando primitivas que implementan algoritmos paralelos conocidos para detectar colisiones múltiples en este grid. Dos de las más comunes son Scan y Reduction [15] como se comentará a continuación.

La primitiva Scan consiste en el cómputo de una suma prefija (prefix sum) de las componentes de un array. Por ejemplo, para el array $[x_0, x_1,..., x_{n-1}]$, el resultado sería $[x_0, (x_0 \oplus x_1), ..., (x_0 \oplus x_1 \oplus ... \oplus x_{n-1})]$. La Tabla 4.3 muestra los valores de entrada y salida para un sencillo ejemplo de Scan.

Tabla 4.3 Primitiva Scan

IN	3	1	7	0	4	1	6	3
OUT	3	4	11	11	15	16	22	25

La primitiva Reduction consiste en reducir el array a un solo elemento dependiendo de una operación. La Tabla 4.4 muestra los valores de entrada y salida para un sencillo ejemplo de Reduction en el que la operación usada para compactar el array ha sido el mínimo.

Tabla 4.4 Primitiva Reduction

IN	3	1	7	8	4	1	6	3
OUT	1							

Para el problema del grid espacio-tiempo, no basta con hacer un scan o una reducción simple del array t_i formado por los valores $EPP_1(t_i)$, $EPP_2(_{ti})$, ..., $EPP_N(t_i)$ para detectar una colisión.

Relativo a esta parte, cabe destacar como se comentará en líneas futuras, que en otros trabajos, como [7], se han venido usando Quadtrees para la detección de colisiones, siendo implementados en GPUs. Si el uso de Quadtrees en GPUs es escalable, éste es objeto de un tema a investigar como trabajo futuro en el ámbito de este trabajo.

5 - Resultados experimentales

Para explicar las implementaciones de los diferentes algoritmos estudiados y sus resultados, se introducen los siguientes conceptos:

Cómputo neto de GPU. Cuando se habla de cómputo neto en GPU es referido al cómputo específico del algoritmo dentro de un kernel. En este caso no se tiene en cuenta el coste que supone la transferencia de datos en memoria (transferencia de datos de entrada y transferencia de datos de salida).

Cómputo bruto de GPU. Este es el cómputo total del algoritmo incluyendo transferencias de memoria. Este será el tiempo que se usará para comparar el rendimiento de un algoritmo en GPU frente a su versión secuencial.

Speedup. En computación paralela, el speedup se refiere a cuánto de rápido es un algoritmo paralelo frente a su correspondiente versión secuencial. Se calcula mediante la siguiente fórmula:

$$S_p = \frac{T_1}{T_p} \qquad (22)$$

T_1 es el tiempo de ejecución del algoritmo secuencial en la CPU.

T_p es el tiempo de ejecución del algoritmo paralelo en la GPU.

En el presente trabajo, este speedup es la medida de referencia en la comparación de algoritmos paralelos frente a sus respectivos secuenciales.

A lo largo del capítulo, las gráficas que se muestran están compuestas de un eje *y* en escala logarítmica en base 10 (útil dado que los datos cubren una amplia gama de valores) y un eje *x* de puntos cuyos valores específicos son 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000 considerados como referencia para la ejecución de cada algoritmo.

5.1 - Algoritmo de conversión de coordenadas

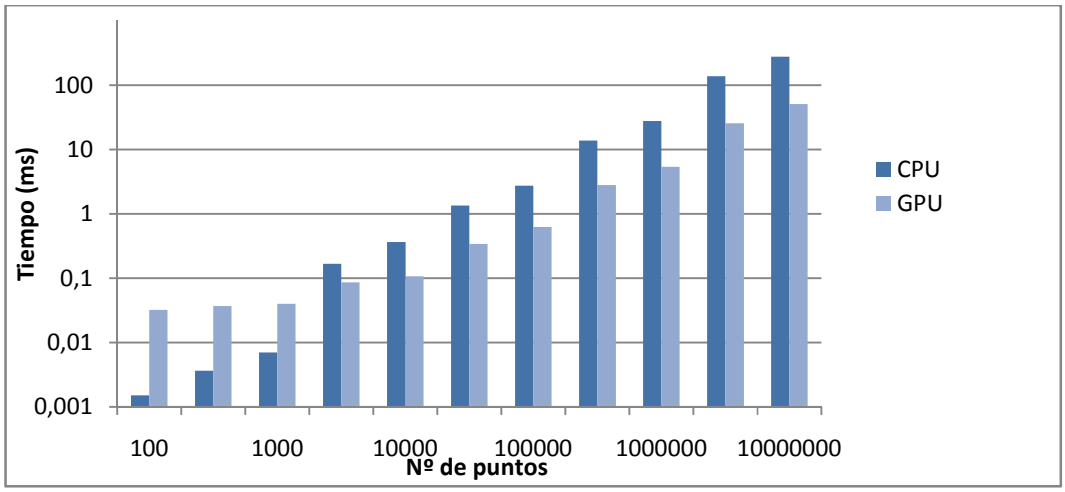
El EPP estándar contiene un máximo de 128 puntos, lo que implica un escaso paralelismo de datos.

El principal problema de paralelizar este algoritmo es que al tiempo de cómputo neto de GPU hay que añadir el tiempo de transferencia de memoria principal de los datos de entrada a la memoria de la tarjeta gráfica y el tiempo de transferencia del resultado de la memoria de la tarjeta gráfica a la memoria principal. Comparando en este caso de pocos puntos, la latencia de memoria es bastante mayor que el tiempo de cómputo del algoritmo de conversión, y por ello no resulta eficiente hacer este cálculo en la GPU.

Debido a este problema, no he realizado otras optimizaciones de uso de memoria en el algoritmo de conversión de coordenadas.

Aunque las trayectorias ordinarias se consideran de 128 puntos, vamos a estudiar el comportamiento del acelerador para la conversión de coordenadas con mayor número de puntos, pues parece indicado pensar que en el futuro la capacidad de la aeronave de calcular su trayectoria de forma realista será incrementar el número de puntos. Bajo este paradigma las siguientes figuras muestran los tiempos de ejecución del algoritmo de conversión de coordenadas para un número variable de puntos.

Figura 5.1 Comparación de tiempos en la conversión de coordenadas

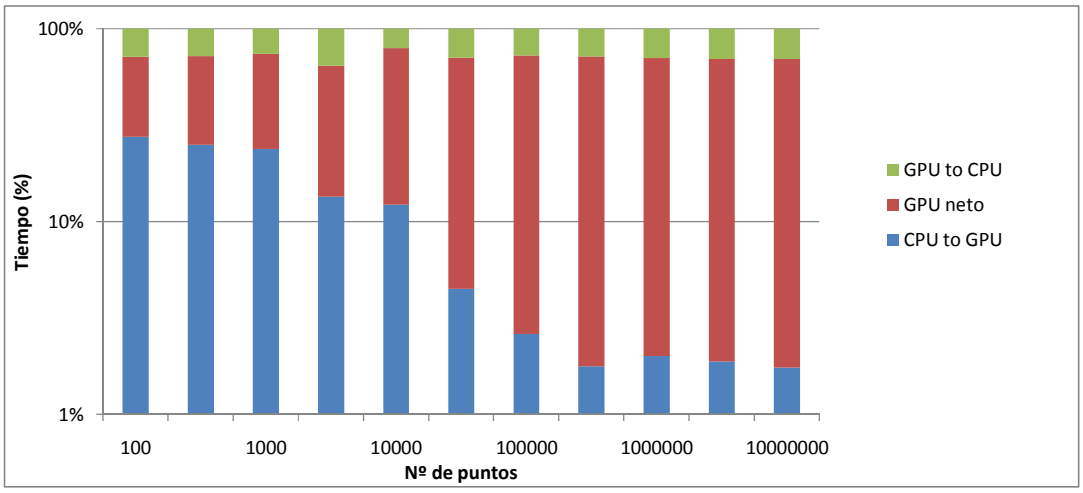


En el anterior gráfico de la Figura 5.1 se compara el tiempo de cómputo en la GPU con respecto al tiempo de CPU. El comportamiento mejora en gran medida para valores de entrada altos, llegando a un speedup pico de 8x.

En la siguiente gráfica, Figura 5.2, se observa la porción de tiempo usada para cada operación de GPU. El tiempo de ejecución del kernel es el que predomina según se incrementa el número de puntos. Se puede observar que el tiempo de transferencia de los datos de entrada a la GPU es el principal problema para un número reducido de puntos. El tiempo de transferencia de la salida es siempre el mismo porque los datos de salida tienen siempre el mismo tamaño.

De acuerdo a la gráfica se sugiere una buena escalabilidad. El cómputo del kernel se acaba llevando más del 70% del tiempo total frente al 30% de la transferencia de memoria.

Figura 5.2 Uso total de tiempo GPU en la conversión de coordenadas

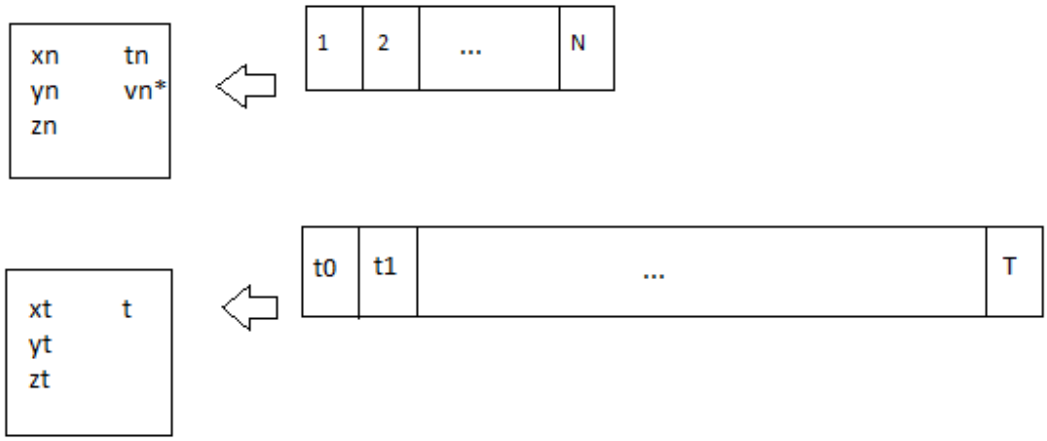


Definitivamente, para nuestro problema concreto de 128 puntos, observamos como la implementación paralela en GPU es menos eficiente que su implementación secuencial en CPU para 128 puntos, aunque se ha comprobado que si el EPP se estandarizara en el futuro con mayor número de puntos, resultaría adecuado hacer la conversión de coordenadas mediante GPUs a partir del punto de inflexión aproximadamente situado en 3000 puntos.

5.2 - Algoritmos de interpolación

Para obtener un muestreo de la trayectoria se han implementado de forma paralela diferentes métodos de interpolación en la CPU y en la GPU, con el fin de medir cuál aporta más ganancia aplicando una serie de optimizaciones y diferenciando además que método es el más apropiado para esta tarea. En los tres casos estudiados se transfiere como entrada un vector de 128 puntos y se obtiene otro vector de tamaño variable que contiene el muestreo de la trayectoria a partir de un intervalo de tiempo fijo. La representación de los datos se observa en la Figura 5.3:

Figura 5.3 Trajectory sampling



donde el vector de entrada contiene N puntos con las coordenadas cartesianas, el tiempo y la velocidad estimada por el EPP y el vector de salida contiene T puntos desde t_0 (tiempo inicial) en intervalos $t_i = t_{i-1}$ + time_step. Cada punto de salida contiene como resultado las coordenadas cartesianas en un instante t como interpolación de los valores x, y, z para $t_{n-1}<t<t_n$.

5.2.1 - Interpolación lineal

La interpolación lineal usa las coordenadas cartesianas de los puntos adyacentes para obtener la posición de la aeronave en un tiempo t, según la fórmula vista en el Capítulo 4. El algoritmo secuencial se muestra en la figura 5.4 de la página siguiente:

Figura 5.4 Código de la interpolación lineal

```
void linear_interpolation(point* orig, point* dest, int length) {

  int t_max = MAX_TIME;
  int t_step = TIME_STEP;
  int t_ini = 1;

  bool term = false;
  point last;

  int index = 0;
  int dest_index = 0;

  int t = t_ini;

  while (t<=t_max) {

      while (orig[index].time < t) {

        if (index + 1 > length) term = true;

        index++;
      }

      if (!term) {

        double x = orig[index - 1].pos.x +
      (t - orig[index - 1].time) *
      (orig[index].pos.x - orig[index - 1].pos.x) /
      (orig[index].time - orig[index - 1].time);

        double y = orig[index - 1].pos.y +
      (t - orig[index - 1].time) *
      (orig[index].pos.y - orig[index - 1].pos.y) /
      (orig[index].time - orig[index - 1].time);

        double z = orig[index - 1].pos.z +
      (t - orig[index - 1].time) *
      (orig[index].pos.z - orig[index - 1].pos.z) /
      (orig[index].time - orig[index - 1].time);


        dest[dest_index] = (point){{x, y, z}, t};
        last = dest[dest_index];

      }
      else {dest[dest_index] = last;}

      t = t + t_step;
      dest_index++;
    }
  }
```

A primera vista se puede observar que la paralelización de la interpolación lineal se puede realizar de forma directa porque cada cálculo es independiente de los demás. En las gráficas de rendimiento se observa como el speedup frente al algoritmo secuencial es considerable.

Las optimizaciones en GPU que han sido implementadas consisten en mejorar la localidad y la latencia de acceso a los datos:

(a) Uso de la memoria compartida.

Cada operación necesita dos datos del vector de entrada orig[i] y orig[i - 1]. Como el número de puntos de salida será mucho mayor que el de entrada, sin duda existen multitud de datos de entrada que serán reutilizados por diversos hilos. Para reducir el tiempo de acceso se han utilizado las siguientes variables

```
__shared__ double xcoord[N];
__shared__ double ycoord[N];
__shared__ double zcoord[N];
```

Se separa el array de puntos en sus arrays de coordenadas. Las primeras veces que se accede a los datos de entrada no se encuentran en zona de memoria compartida, falla el acceso y el acceso se realiza a memoria principal. Una vez se van reutilizando datos, los accesos van dejando de fallar y los accesos empiezan a ser menos costosos (la memoria compartida está en el chip).

Se observa que esta optimización depende bastante de la relación del tamaño de datos de entrada (N) y de salida (T), a mayor tasa de muestreo, mejor se comporta esta optimización:

El speedup mejora ligeramente cuando $\frac{T}{N} \geq 6$, aunque parece que esta mejora se mantiene siendo efectiva hasta valores de $\frac{T}{N} = 80$, como se puede apreciar en la gráfica de rendimiento.

(b) Uso de memoria constante o de memoria de texturas.

Estos espacios se encuentran fuera del chip pero están cacheados, por lo que se ha intentado hacer lo mismo que con la memoria compartida aunque con peores resultados. Puede ser debido a que el tamaño de los datos de entrada es bastante grande con respecto al tamaño de la memoria constante.

(c) Distribución de carga uniforme entre bloques e hilos.

Una buena configuración para lanzar el kernel es:

```
int threadsPerBlock = 256;
int blocksPerGrid = (T + threadsPerBlock - 1) / threadsPerBlock;

LinearInterpolation<<<blocksPerGrid, threadsPerBlock>>>(orig, dest, T);
```

El índice de acceso dentro del hilo no se establece de forma secuencial:

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

Como se necesitan intervalos de tiempo bastante reducidos para después desarrollar la comparación, se obtiene la mayor cantidad de muestras de la trayectoria y los resultados son muy positivos (speedup pico 100x), como se observa en las Figuras 5.5 y 5.6:

Figura 5.5 Speedup de la interpolación lineal

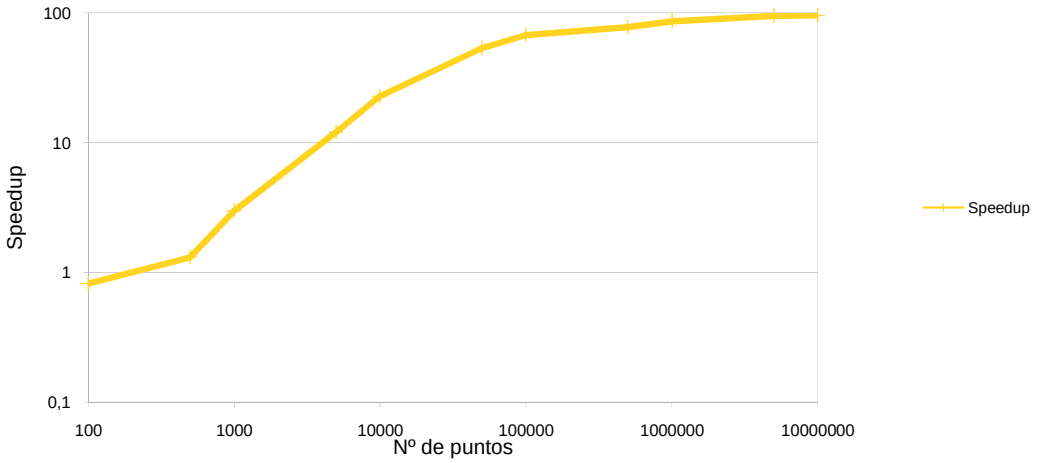
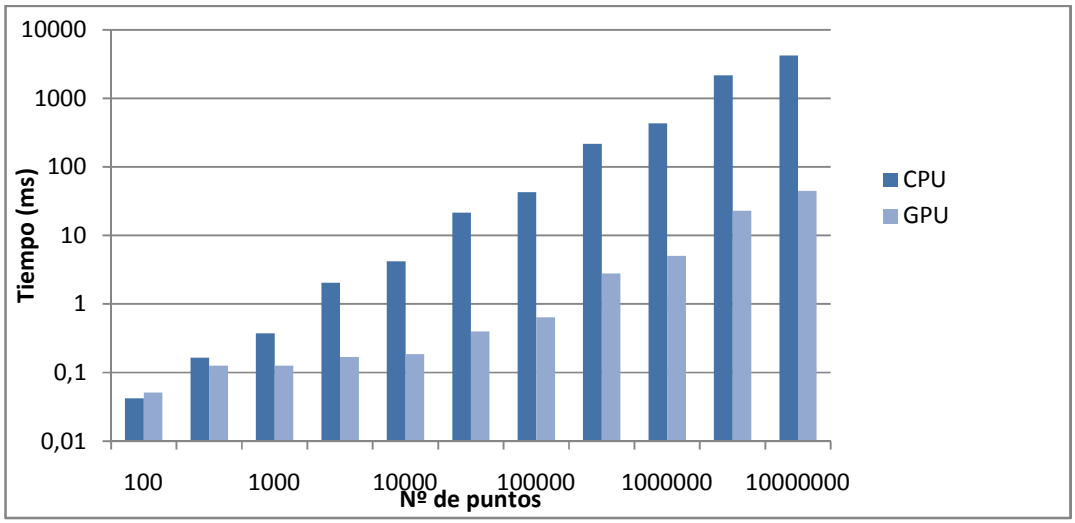


Figura 5.6 Comparación temporal de la interpolación lineal en GPU y CPU



Si bien es cierto que no se va a necesitar una interpolación lineal de 10 millones de puntos para conseguir una comparación de trayectoria completa, el speedup conseguido para este algoritmo es dos órdenes de magnitud. En la práctica, para intervalos en segundos de unas 4 horas de vuelo, el speedup frente a la implementación secuencial se encuentra entre 30x y 40x.

5.2.2 - Interpolación cuadrática

Con la interpolación lineal paralela hemos obtenido buenos resultados de rendimiento, pero para la interpolación cuadrática se ha obtenido un rendimiento similar con la particularidad de que los puntos interpolados son de mayor calidad. Al introducir la velocidad en el cálculo, el número de operaciones en cada hilo es bastante mayor.

Debemos realizar cálculos intermedios para los coeficientes expresados en el Capítulo 4, pero las optimizaciones del algoritmo son básicamente las mismas que el apartado anterior. La implementación de dichos coeficientes se muestra en la Figura 5.7 a continuación:

Figura 5.7 Coeficientes calculados para la interpolación cuadrática

```
double last_alpha = fabs(atan2(orig[index].pos.y - orig[index - 1].pos.y,
                               orig[index].pos.x - orig[index - 1].pos.x));
double vx_last = fabs(orig[index - 1].v * cos(last_alpha));
double vy_last = fabs(orig[index - 1].v * sin(last_alpha));

double next_alpha = fabs(atan2(orig[index + 1].pos.y - orig[index].pos.y,
                               orig[index + 1].pos.x - orig[index].pos.x));
double vx_next = fabs(orig[index - 1].v * cos(next_alpha));
double vy_next = fabs(orig[index - 1].v * sin(next_alpha));


double A = (vx_last - vx_next) *
           pow((orig[index].time - t), 2) +
           2 * vx_next * (orig[index].time - orig[index - 1].time) *
           (orig[index].time - t);

double B = (vx_next - vx_last) *
           pow((t - orig[index - 1].time), 2) +
           2 * vx_last * (orig[index].time - orig[index - 1].time) *
           (t - orig[index - 1].time);

double C = (vx_last + vx_next) *
          pow((orig[index].time - orig[index - 1].time), 2);

double D = (vy_last - vy_next) *
          pow((orig[index].time - t), 2) +
          2 * vy_next * (orig[index].time - orig[index - 1].time) *
          (orig[index].time - t);

double E = (vy_next - vy_last) *
          pow((t - orig[index - 1].time), 2) +
          2 * vy_last * (orig[index].time - orig[index - 1].time) *
          (t - orig[index - 1].time);

double F = (vy_last + vy_next) *
          pow((orig[index].time - orig[index - 1].time), 2);
```

El speedup se mantiene con respecto a la interpolación lineal (CPU vs GPU). Los tiempos de ejecución son algo superiores, pero con este método las coordenadas obtenidas son bastante más reales y muy parecidas a la trayectoria recibida en el EPP desde la aeronave.

5.2.3 - Interpolación polinómica

En este caso práctico se toma la interpolación polinómica cúbica por splines para cada una de las coordenadas (x_i,y_i,z_i) definida en el capítulo anterior. Las sentencias para interpolar cada una de las coordenadas están formadas por operaciones aritméticas, implementadas según la Figura 5.8 de la siguiente página:

Figura 5.8 Coeficientes y valores calculados para la interpolación por splines

```
float   ax, bx, cx;
float   ay, by, cy;
float   az, bz, cz;
float   tSquared, tCubed;

/* cálculo de los coeficientes polinomiales */
cx = 3.0 * (firstControlPoints[index].pos.x - orig[index - 1].pos.x);
bx = 3.0 * (secondControlPoints[index].pos.x - firstControlPoints[index].pos.x) - cx;
ax = orig[index].pos.x - orig[index - 1].pos.x - cx - bx;

cy = 3.0 * (firstControlPoints[index].pos.y - orig[index - 1].pos.y);
by = 3.0 * (secondControlPoints[index].pos.y - firstControlPoints[index].pos.y) - cy;
ay = orig[index].pos.y - orig[index - 1].pos.y - cy - by;

cz = 3.0 * (firstControlPoints[index].pos.z - orig[index - 1].pos.z);
bz = 3.0 * (secondControlPoints[index].pos.z - firstControlPoints[index].pos.z) - cz;
az = orig[index].pos.z - orig[index - 1].pos.z - cz - bz;

float t_interval = (100.0 - (100.0 *(orig[index].time - t)/
                                    (orig[index].time - orig[index - 1].time)))/100.0;

tSquared = t_interval * t_interval;
tCubed = tSquared * t_interval;

double x = (ax * tCubed) + (bx * tSquared) + (cx * t_interval) + orig[index - 1].pos.x;
double y = (ay * tCubed) + (by * tSquared) + (cy * t_interval) + orig[index - 1].pos.y;
double z = (az * tCubed) + (bz * tSquared) + (cz * t_interval) + orig[index - 1].pos.z;
```

La principal diferencia en la implementación del kernel con respecto a las interpolaciones lineal y cuadrática vistas anteriormente es que se necesitan dos puntos de control para cada intervalo. Como este array (array de puntos de control) es calculado dentro del kernel se usa la memoria compartida para aprovechar cálculos entre hilos. En este caso si es importante resaltar que habrá bastantes hilos que puedan usar los puntos de control calculados por otro hilo anteriormente, por lo que cabe esperar que la optimización puedan ser válida. Para facilitar la comprensión, se usan dos arrays de puntos de control:

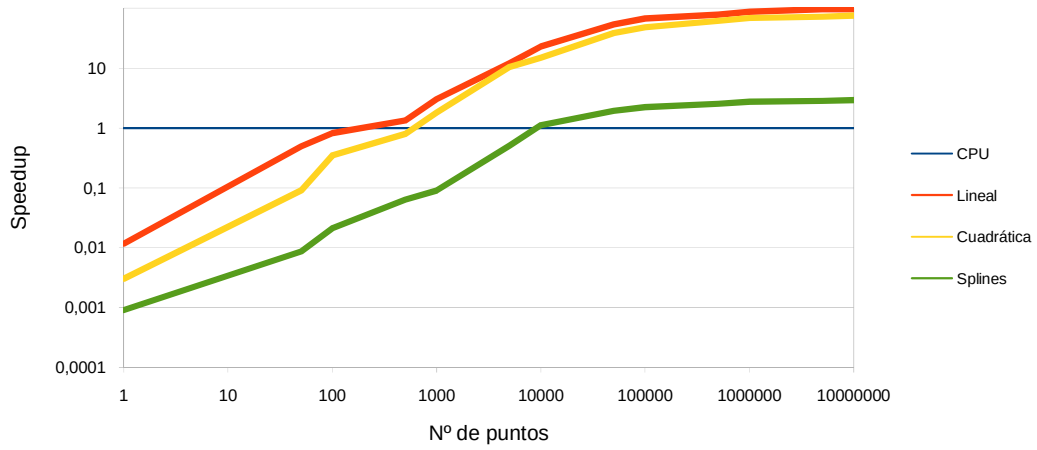
```
__shared__ point firstControlPoints[N-1];
__shared__ point secondControlPoints[N-1];
```

Aquí se vuelve a remarcar una línea para el trabajo futuro a estudiar, como una mejora de la distribución de dichos puntos de control, puesto que el algoritmo que los calcula tiene una

fuerte componente secuencial que provoca divergencia en los hilos. Cuando los hilos que se ejecutan a la vez toman caminos distintos (las sentencias condicionales hacen que en cada hilo se ejecuten instrucciones distintas), dichos caminos diferentes se serializan, penalizando en gran medida la componente paralela del algoritmo.

En este caso, se obtiene un speedup de 3x con respecto a la versión secuencial cuando la cantidad de puntos a interpolar es alta. Sin embargo, el tiempo de ejecución aumenta considerablemente con respecto a las interpolaciones anteriormente implementadas en paralelo, con el añadido de que el objetivo de éste método era conseguir una baja tasa de muestreo con mayor calidad de puntos interpolados. El comportamiento para una baja tasa de muestreo es poco eficiente. Además, la calidad de los puntos conseguidos no mejora en gran medida a la de los obtenidos por interpolación cuadrática, debido con seguridad a que los puntos recibidos por medio del EPP contienen los puntos más significativos de la trayectoria de la aeronave. Además, no ha sido usada la velocidad estimada para el cálculo a diferencia de la interpolación cuadrática.

Figura 5.9 Comparativa de speedup para los métodos de interpolación



5.3 - Algoritmo de detección de conflictos

5.3.1 - Detección geométrica

La estrategia para la detección geométrica usada parte de dos trayectorias obtenidas mediante interpolación cuadrática con alta tasa de muestreo. Debido a eso, el algoritmo paralelo de comparación es directo.

La razón de elegir la interpolación cuadrática es debida a un pequeño estudio de precisión que tiene en cuenta los siguientes parámetros:

- Número de puntos necesarios para obtener suficiente ganancia con respecto a la implementación en CPU.
- Tiempo de ejecución parcial de la interpolación.
- Calidad de los puntos obtenidos.
- Distancia de separación necesaria.

Tabla 5.1 Estudio de precisión para los métodos de interpolación

Interpolación	**Lineal**	**Cuadrática**	**Polinómica**
Nº de puntos	+++	+++	+
Tiempo de ejecución	++++	++++	+
Calidad	++	+++	+++
Distancia de separación	++	+++	++

Tanto la interpolación lineal como la cuadrática obtienen mejor rendimiento en GPU con respecto a su versión en CPU. El tiempo de ejecución de ambas es bastante más reducido que el de la interpolación polinómica, como indica la Tabla 5.1.

Por otra parte, la calidad de los puntos obtenidos es algo menor en el caso de la interpolación lineal, forzando a que la distancia de separación entre aeronaves tenga que ser mayor que en el caso de la interpolación cuadrática.

Se puede decir que la distancia de separación μ es inversamente proporcional al número de puntos de trayectoria N e inversamente proporcional al *Figure of Merit Level (FoM)* visto en

el capítulo anterior. Se mantendrá una distancia de separación mínima μ_{min} impuesta externamente (dependiente de la estela turbulenta, los datos meteorológicos, etc.) y se añade otra constante (Q) que indica la calidad del método de interpolación:

$$\mu = \mu_{min} + \frac{Q}{N * FoM} \qquad (23)$$

Si se optara por tener como entrada las trayectorias con menor tasa de muestreo obtenidas con la interpolación polinómica tendríamos que usar otro algoritmo que tuviera en cuenta puntos adyacentes. Este último punto puede ser considerado como trabajo futuro que podría resultar interesante investigar si se mejoran los resultados de la interpolación por splines.

El cálculo geométrico de distancia 3D entre puntos que se puede hacer es sencillo debido al muestreo y a la conversión de coordenadas geodésicas a cartesianas. Otro autores, como [9], se ven forzados a simplificar al cálculo de distancia 2D puesto que deben incluir en la fórmula otras variables como el tiempo y detalles de conversión. El algoritmo de detección se simplifica a una comparación punto a punto sin necesidad de uso de la memoria compartida, debido al muestreo realizado anteriormente.

Usamos la ecuación de distancia entre dos puntos en cada instante t bajo una serie de restricciones que conforman la distancia de seguridad vertical y horizontal de la aeronave, cuya implementación muestra la Figura 5.10.

Figura 5.10 Instrucción principal del kernel de detección de conflictos

```
C[index] = sqrt(pow(B[index].pos.x - A[index].pos.x, 2) +
                pow(B[index].pos.y - A[index].pos.y, 2) +
                pow(B[index].pos.z - A[index].pos.z, 2)) < sep_distance
```

En este caso, como cada hilo accede a unos únicos valores de cada trayectoria, no sería acertado usar la memoria compartida para adelantar datos, puesto que no habrá reuso (localidad temporal).

Por otra parte, la ejecución de este kernel nunca provocará divergencia entre los hilos. Debido al algoritmo de interpolación cuadrática con alta tasa de muestreo implementado en la etapa anterior hemos conseguido que la comparación sea un problema SIMT (Single Instruction, Multiple Threads) óptimo para su implementación paralelizable, sin necesidad de usar instrucciones condicionales dentro del kernel. Como se observa en la Figura 5.11, se obtiene el siguiente speedup con respecto a una versión secuencial, y cuyos tiempos de ejecución son los de la Figura 5.12:

Figura 5.11 Speedup en el algoritmo de detección geográfica

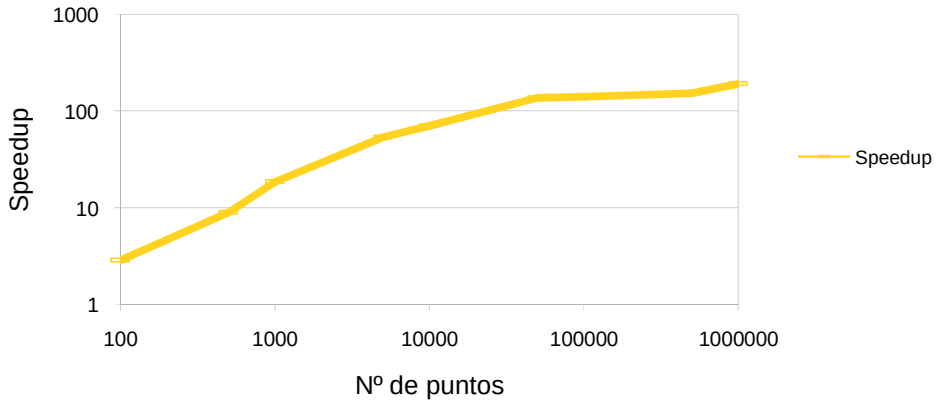
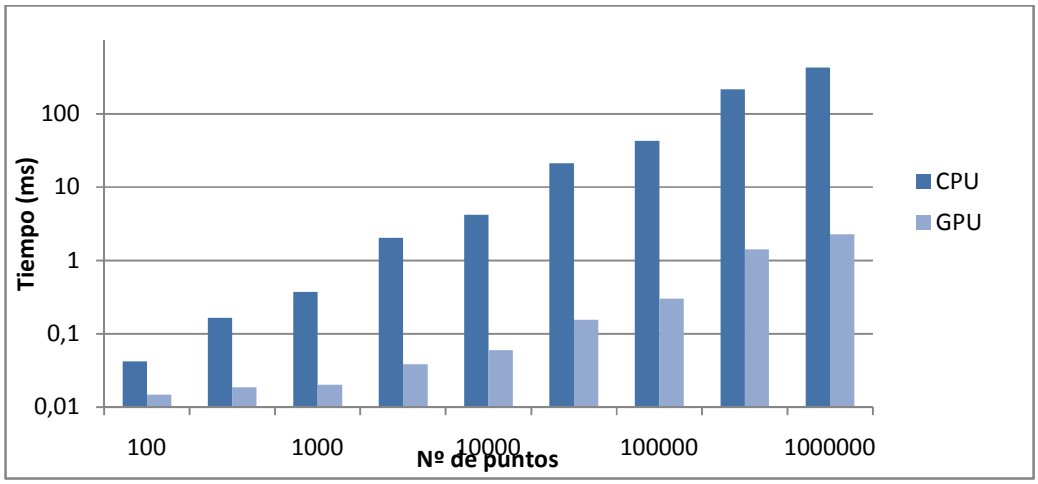


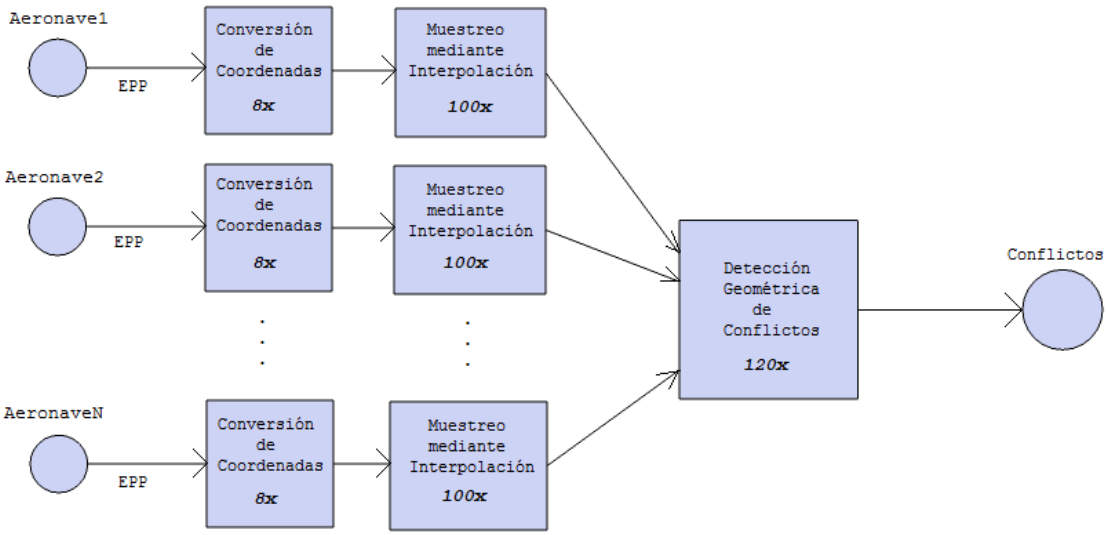
Figura 5.12 Tiempo CPU vs GPU para la de detección geográfica de conflictos



5.4 - Análisis de rendimiento

Recuperando la propuesta del capítulo 4, y su correspondiente gráfica de la Figura 4.1 dónde se presentaba la división por etapas que se ha implementado, vemos a continuación en la Figura 5.13 el throughput global del sistema con la acumulación de ganancia en tiempo de la solución GPU frente a la CPU:

Figura 5.13 Speedup en Etapas paralelizables de la Comparación de Trayectorias



En las Tablas 5.2 y 5.3 de la siguiente página se pueden observar los tiempos de ejecución de cada etapa. Dependiendo del número de puntos de entrada, se pueden comparar también los tiempos de ejecución total en cada caso para las implementaciones en GPU y en CPU respectivamente. Para un número pequeño de puntos de entrada, inferior a 500, el desarrollo en GPU no resulta más conveniente que la solución secuencial en CPU. Este resultado era de esperar observando la tendencia de cada una de las etapas a ir mejorando tiempos según se incrementaba el número de puntos. El algoritmo paralelo de Comparación de Trayectorias completo implementado en GPU es más eficiente en tiempo cuanto mayor número de puntos contiene el EPP obtenido de la aeronave y mejora bastante los tiempos de ejecución en CPU.

Tabla 5.2 Resumen de tiempos en la implementación GPU (ms)

Nº Puntos	Transferencia CPU to GPU	Conversión de Coordenadas	Interpolación Cuadrática	Detección Geométrica	Transferencia GPU to CPU	Total
100	0,009	0,014	0,051	0,015	0,009	0,098
500	0,009	0,017	0,126	0,019	0,010	0,181
1000	0,010	0,020	0,126	0,020	0,010	0,186
5000	0,012	0,044	0,169	0,038	0,031	0,294
10000	0,013	0,072	0,185	0,060	0,023	0,352
50000	0,015	0,225	0,399	0,155	0,100	0,895
100000	0,016	0,436	0,635	0,302	0,174	1,563
500000	0,050	1,958	2,780	1,417	0,787	6,992
1000000	0,108	3,688	5,003	2,267	1,611	12,677

Tabla 5.3 Resumen de tiempos en la implementación CPU (ms)

Nº Puntos	Conversión de Coordenadas	Interpolación Cuadrática	Detección Geométrica	Total
100	0,002	0,042	0,010	0,054
500	0,004	0,165	0,083	0,251
1000	0,007	0,375	0,092	0,474
5000	0,168	2,037	0,343	2,548
10000	0,365	4,205	1,792	6,362
50000	1,337	21,359	12,078	34,774
100000	2,754	42,719	22,666	68,139
500000	13,805	216,621	132,267	362,693
1000000	27,920	432,776	261,121	721,817

6 - Conclusiones y trabajo futuro

Cuando un problema se puede descomponer hasta convertirlo en subtareas independientes que ejecutan una misma instrucción para cada uno de los datos de entrada, la capacidad de ejecutar rápidamente cálculos intensivos de la GPUs hace que el resultado de esta descomposición sea por norma general más eficiente que la versión secuencial del algoritmo. En el caso de la comparación de trayectorias, fruto de una necesidad real que nace en los actuales sistemas de Control de Tráfico Aéreo, el uso de GPUs parece altamente recomendable. La escalabilidad, el bajo coste y el mínimo periodo de aprendizaje inicial de CUDA hacen que sea una opción interesante a tener en cuenta para múltiples desarrollos. La tarjeta gráfica empleada, NVIDIA GeForce GTX660, es adecuada para este tipo de problema puesto que el número de procesadores y la elevada tasa de transferencia de memoria juegan un papel fundamental para obtener el rendimiento de cálculo esperado. Cabe destacar también la capacidad de aritmética de doble precisión de esta tarjeta, cumpliendo la necesidad que se tenía al inicio de este estudio, y que ha sido usada en la definición de coordenadas para todas las etapas descritas.

Este estudio remarca que el uso coherente de la memoria es el área donde se obtiene una mayor diferencia de rendimiento de GPU después de la división en partes paralelizables del problema, que principalmente se ha enfocado en el tiempo de acceso a los datos. Dicho eso, la principal aportación ha sido idear y diferenciar las etapas que transforman el problema a una versión paralelizable, y por supuesto valorar si las actuales GPUs son capaces de mejorar algoritmos con esta naturaleza.

El objetivo de conseguir un algoritmo que responda en tiempo real al problema ha sido satisfecho, concatenando los diferentes algoritmos que hemos explicado y que han sido seleccionados tratando de primar el rendimiento y la simplicidad para conseguir una fuerte base a desarrollar en el futuro.

Trabajo futuro

Como trabajo futuro quedan principalmente dos tareas. Por un lado, generalizar la comparación geométrica entre dos aeronaves a N de forma distribuida mediante un grid espacio-tiempo apropiado que pueda ser optimizado usando Quadtrees o distintos métodos ya evaluados

por la comunidad científica. Por otra parte, complementar estos resultados con otros métodos y otras arquitecturas que cubran las situaciones para las que la paralelización pueda llegar a ser no adecuada, o necesite de otras cualidades que no perjudiquen el rendimiento en GPUs, como por ejemplo, debido a que no sea necesario muestrear las trayectorias para detectar conflictos tempranos. En este contexto, la posibilidad de utilizar un acelerador tipo Intel-Xeon-Phi reusando parte del código implementado en este trabajo podría ser muy recomendable.

Cabe también la posibilidad de estudiar en el futuro otras aproximaciones suponiendo que las trayectorias recibidas en el EPP puedan ser complementadas con métodos tradicionales de predicción de trayectorias, constituyendo sistemas más complejos e incluso colaborativos entre distintos Centros de Control Aéreo.

Referencias

[1] High Level Group on Aviation Research. Flightpath 2050 Europe´s Vision for Aviation. European Commision, 2011.

[2] Tandale M. D., Wiraatmadja S., Menon P. K., and Rios J. L., "High-Speed Prediction of Air Traffic for Real-Time Decision Support," AIAA Guidance Navigation and Control Conference, Portland OR, 8-11 August, 2011.

[3] Torres, S.; Klooster, Joel K.; Liling Ren; Castillo-Effen, M., "Trajectory Synchronization between air and ground trajectory predictors," Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th , vol., no., pp.1E3-1,1E3-14, 16-20 Oct. 2011.

[4] Sancı S., İşler V., "A Parallel Algorithm for UAV Flight Route Planning on GPU," International Journal of Parallel Programming, Volume 39, Issue 6, pp 809-837, December 2011.

[5] Weber, R.; Cruck, E., "Subliminal ATC Utilizing 4D Trajectory Negotiation," Integrated Communications, Navigation and Surveillance Conference, 2007. ICNS '07 , vol., no., pp.1,9, April 30 2007-May 3 2007.

[6] Chan, D.S.K.; Brooksby, G.W.; Hochwarth, J.; Klooster, Joel K.; Torres, S., "Air-ground trajectory synchronization — Metrics and simulation results," Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th , vol., no., pp.1E1-1,1E1-14, 16-20 Oct. 2011.

[7] I. Cabañas Ruiz, Construcción de estructuras espaciales en GPU, Proyecto Fin de Máster Universidad Complutense, 2012.

[8] Mejia, J.S.; Stipanovic, D.M., "Safe trajectory tracking for the two-aircraft system," Electro/Information Technology, 2007 IEEE International Conference on , vol., no., pp.362,367, 17-20 May 2007.

[9] Yokoyama, N., "Decentralized algorithm applicable to parallel trajectory planning of multiple aircraft," Control, Automation and Systems, 2007. ICCAS '07. International Conference on , vol., no., pp.1393,1398, 17-20 Oct. 2007.

[10] M.M. Paglione, H.F.Ryan.Trajectory Prediction Accuracy Report U.S. Deparment of Transportation ,1999.

[11] Jamrok, R., "Aircraft datalink communications for the future," Digital Avionics Systems, 2001. DASC. 20th Conference , vol.2, no., pp.6A6/1,6A6/9 vol.2, Oct 2001.

[12] Teutsch, J.; Hoffman, E., "Aircraft in the future ATM system - exploiting the 4D aircraft trajectory," Digital Avionics Systems Conference, 2004. DASC 04. The 23rd , vol.1, no., pp.3.B.2,31-22 Vol.1, 24-28 Oct. 2004.

[13] F. J. Martín Crespo, The Collision Avoidance Problem: Methods and Algorithms, Universidad Rey Juan Carlos, 2010.

[14] Zhang Zhong; Zhang Xuejun, "An improved geometric approach to conflict resolution in curve trajectory," Electronic Measurement & Instruments, 2009. ICEMI '09. 9th International Conference on , vol., no., pp.1-220,1-224, 16-19 Aug. 2009.

[15] NVIDIA CUDA C Best Practices Guide, 2012.

[16] NVIDIA CUDA C Programming Guide, 2011.

[17] TOP500 Supercomputer Sites, www.top500.org.

[18] Crow, R., "Aviation's next generation global CNS/ATM system," Position Location and Navigation Symposium, 2002 IEEE , vol., no., pp.291,298, 2002.

[19] Ji-wung Choi; Curry, R.; Elkaim, G., "Path Planning Based on Bézier Curve for Autonomous Ground Vehicles," World Congress on Engineering and Computer Science 2008, WCECS '08. Advances in Electrical and Electronics Engineering - IAENG Special Edition of the , vol., no., pp.158,166, 22-24 Oct. 2008.